HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

# Programmering i gymnasiets långa matematik enligt läroplanen för år 2021

Helsingfors universitet
Matematisk-naturvetenskapliga fakulteten
Institutionen för matematik och statistik
Lärarutbildningen
Pro gradu -avhandling
Matematik
Februari 2020
Tom Olander

Handledare: Matti Pauna

| Tiedekunta - Fakultet - Faculty | Laitos - Institution - Department |
|---|---|
| Matematisk-naturvetenskapliga fakulteten | Institutionen för matematik och statistik |

Tekijä - Författare - Author
Tom Olander

Työn nimi - Arbetets titel
Programmering i gymnasiets långa matematik enligt läroplanen för år 2021

Title

Oppiaine - Läroämne - Subject
Matematik

| Työn laji/ Ohjaaja - Arbetets art/Handledare - Level/Instructor | Aika - Datum - Month and year | Sivumäärä - Sidoantal - Number of pages |
|---|---|---|
| Pro gradu-avhandling / Matti Pauna | 12.2.2020 | 56 sidor |

Tiivistelmä - Referat - Abstract

Denna avhandling ställer frågor kring vad det innebär för den långa matematiken i gymnasiet när programmering tillkommer enligt läroplanen. Först granskas den tidigare forskningen kring programmering och matematik i gymnasiet och till vilka slutsatser man kommit i dessa. Att programmering kan vara till nytta för studeranden i yrkeslivet är givet, men huruvida programmering skall höra till matematiken är en av huvudfrågorna i avhandlingen.

Eftersom det inte finns mycket forskning kring programmering i matematik i Finland har här även används forskning gjort i andra länder. Samma sak gäller för den aktuella åldersgruppen, därför har även forskning med studeranden i ungefär samma ålder använts.

Inget undervisningsmaterial ännu finns för programmeringen i gymnasiet. Därför finns här också förslag till uppgiftstyper som kunde användas i undervisningen. Dessa exempel får fritt modifieras och användas som hjälp vid planering av undervisningen. En annan användning av denna avhandling kunde vara att låta den vara som grund för kommande planering av programmering i gymnasiet angående så väl i vilka ämnen programmering hanteras och hur denna undervisning skall ske i avseende av programmeringsspråk och kontrollering av kodens funktion.

Avainsanat - Nyckelord
Programmering, matematik, gymnasiet

Keywords

Säilytyspaikka - Förvaringsställe - Where deposited
Campus Gumtäkts bibliotek

Muita tietoja - Övriga uppgifter - Additional information

# Innehåll

# Figurer



# Tabeller

# Kodexempel

# Inledning

Jag börjar denna text med att berätta min egen backgrund i fråga om programmering och matematik. Jag har jobbat i ca 20 år som lärare i matematik och samtidigt också undervisat i datakurser för högstadiet, däribland också programmering. Därtill har jag fungerat som ”digi-tutor” för lågstadieskolor ett läsår och då var det huvudsakligen programmering som jag hjälpte till lärarna med i klass situation. Jag kommer i texten även att hänvisa till mina egna erfarenheter inom läraryrket.

Enligt den nya läroplanen som börjar tillämpas i gymnasierna i Finland år 2021 skall man ta med programmering som en del av den långa matematiken. Redan tidigare har programmering varit en del av matematiken också i grundskolan, så denna utvekling var ingen överraskning.

Det är ett känt faktum att matematiker tillämpar programmering i deras olika yrken och därför kan även denna utveckling inom skolan verka som en bra plan. Det har visat sig svårt att i forskningen påvisa huruvida programmering hjälper studerande att lära sig matematik, man antar att vi kanske inte ännu hittat de rätta sätten att undersöka denna sak med. Programmering utvecklar det logiska tänkandet, men matematik i skolan kan också vara mycket mera än endast logiskt tänkande. Matematiken i grundskolan och gymnasiet går mera ut på att studerandena skall lära sig allmänna begrepp och metoder inom matematik, inte att de nödvändigtvis alltid förstår hur och varför vissa formler har kommit till. För gemene man är det viktigare att kunna tillämpa matematiken än att förstå i minsta detalj beviset eller härledningen för de tillämpade formlerna. Det vore till fördel om studerandena förstod bevisen till formlerna och också kunna härleda de allmännaste formlerna på egen hand, men det finns inte alltid tillräckligt med tid för detta i skolan. Denna fördjupade kunskap får studerande som fortsätter att studera matematiska ämnen i universitet eller tekniska högskolor. De mest grundläggande formlerna bör naturligtvis härledas för de studerande så att de också blir bekanta med hur en matematisk härledning sker, rigorositeten i härledningen bör vara på den nivån studeranden kan hantera.

Härmed kommer man till den springande punkten. Programmering kan gynna det logiska tänkandet, men på bekostnad av tid från undervisning i egentlig matematik. Programmering kunde gärna få vara en del av studieutbudet i gymnasieutbildningen, men då borde den inte vara bunden enbart till den långa matematiken, även studeranden som valt kort matematik borde få ta del. Därför borde programmeringen i gymnasiet vara en egen kurs, och för att den skall finnas med i alla gymnasiers läroplaner borde kursen vara obligatorisk. I andra fall bibehålls ”status quo”, de gymnasier som redan tidigare haft programmering som tillvalskurser fortsätter med dem, medan de övriga skolorna endast tar med programmering som valbara kurser – om det finns lämpliga lärare för detta i den aktuella lärarkåren. I en del europeiska länder har man redan tagit med programmering som ett eget undervisningsämne (SchoolNet).

## Programmering i grundskolan och gymnasiet

Eftersom programmeringen i gymnasiet kan antas vara en fortsättning på programmeringen i grundskolan behandlas här även vissa aspekter angående hur programmeringen hanteras i grundskolan.

Också i grundskolan är programmeringen bunden till matematiken, även om ämnesövergripande studier förväntas. Detta gör att lärarna säkerligen kommer att försöka skapa sådana programmeringsuppgifter, eller använda uppgifter som kommer att finnas i de kommande läromedlen, som har en någorlunda stark koppling till matematiken. Detta för att kunna få en del av tiden som används till programmering till nytta för undervisningen i matematik. Detta tankesätt är förståeligt och naturligt, däremot kan detta visa sig svårt att tillämpa ibland, då inte alla delar i matematiken alltid är lika lämpade för överföring till programmering. Detta försvåras ytterligare av det faktum att de flesta eleverna i grundskolan är nybörjare inom programmering.

Då det är allmänt känt att elever ofta har lättare att lära sig nya saker om det sker visuellt torde programmeringen också kunna dra nytta av detta. På 1980-talet när programmering för första gången infördes i skolor använde man sig ofta av programmeringsspråk som innehöll grafiska verktyg för att enkelt kunna rita geometriska objekt på skärmen, ofta var det frågan om så kallad ”turtle”-grafik. ”Turtle”-grafiken gick ut på att man hade en sköldpadda, eller något annat objekt, som man genom programmeringskod kunde flytta på och även välja ritläge, pennan upp eller ner. Första programmeringsspråket som tillämpade ”turtle”-grafik var LOGO, idag kan

”turtle”-grafik tillämpas på ett eller annat sätt i de flesta allmänna programmeringsspråken.
”Turtle”-grafiken är lämplig för absoluta nybörjare, men har också sin plats för dem som hunnit längre inom programmering, exempelvis kan man enkelt med ”turtle”-grafik visa att en regelbunden månghörning likar mer och mer en cirkel då hörnens antal i månghörningen ökas. Detta är någonting som kan vara av stor nytta för eleverna att märka redan i ett tidigt skede, även om de inte ännu förstår sig på matematiska begrepp som gränsvärde och oändlighet.
Det största problemet med programmering i grundskolan, och därmed också i gymnasiet som en följd där av, är att det inte finns en egentlig plan för hur programmeringen skall genomföras i skolan. Grundskolans läroplan borde vara mera exakt definierad angående hur programmeringen skall undervisas samt hur mycket tid som skall stå tillhanda för detta ändamål. Som det är i dagens läge får skolorna själv välja vad som de går igenom och på vilken nivå samt vilket programmeringsspråk de använder.
Då inte en noggrann plan finns för grundskolans del kan inte heller gymnasiet förutsätta förkunskaper inom programmering för studerandes del och måste därför börja på nytt från grunderna.

Vissa av lärandemålen och de centrala begreppen angående programmering i den nya läroplanen är samtidigt obligatoriska förkunskaper för att kunna utföra programmering, så som logik, algoritmisk tankeförmåga samt delvis talteori. Av denna orsak hanteras även dessa begrepp i denna text, även om de inte direkt har med själva programmeringen att göra.

Den nya läroplanen är ännu så ny att inga undervisningsmaterial kan ligga som grund för denna text. Där av beslutet att här gå igenom olika delar av programmering som kan nyttjas som material i kursens undervisning. De givna lösningarna till de olika delarna är oftast endast ett av många olika sätt att lösa problemet, modellösningarna är utförligt kommenterade för att det skall vara lättare att förstå koden. Val av uppgiftstyper är baserat på kursens övriga centrala innehåll.

# Forskningsunderlaget

## Läroplans grunder för gymnasieutbildning 2021

I grunderna för gymnasiets läroplan för år 2021 (Utbildningsstyrelsen 2019) ställs följande mål och centrala innehåll för kursen MAA11, Algoritmer och talteori, i långa matematiken:

Målen för modulen är att den studerande ska

- veta vad en algoritm är, samt lära sig att undersöka hur en algoritm fungerar
- lära sig att utföra enkla algoritmer genom programmering
- bli insatt i logikens begrepp
- behärska grundbegrepp i talteorin och göra sig förtrogen med primtalens egenskaper
- kunna undersöka delbarheten hos hela tal.

Centralt innehåll

- grundbegrepp i algoritmiskt tänkande: sekvens, val och upprepning
- flödesdiagram
- programmering av enkla algoritmer, sorteringsalgoritmer eller en algoritm som anknyter till numerisk lösning av en ekvation
- konnektiv och sanningsvärden
- delbarhet hos hela tal, delbarhetsekvationen (delningsekvationen) och kongruens
- Euklides algoritm
- aritmetikens grundsats

I denna text skall vi försöka binda så mycket av det centrala innehållet i kursen som möjligt till programmering, som enligt grunderna också skall ingå i kursen. Programmering skall endast vara en lösningsmetod för vissa typer av matematiska problem och därför kommer inte heller allt innehåll i kursen att behandlas här.

## Programmering i grundskolor och gymnasier i Europa

Balanskat och Engelhardt (2015) jämförde år 2015 hur långt de olika länderna i Europa kommit med programmering i skolorna, vilka faktorer som driver länderna i denna fråga och hur man förverkligat undervisningen. Undersökningen omfattar 21 länder, vid detta tillfälle hade man i Finland inte ännu tagit med programmering i läroplanen för grundskolan, men är trots det med i undersökningen. Av de deltagande länderna var det endast 3 länder som inte ännu hade programmering i läroplanen och heller inga planer för detta. Som orsak att ha med programmering i läroplanerna tas främst upp att främja logisk tankegång, problemlösning och programmeringskunskaper. Dessa var även Finlands motiveringar, men flera länder satte också vikt vid fortsatta studier inom datateknik och -vetetenskap samt anställningsmöjligheter. 12 av länderna som deltog i undersökningen hade programmering som ett eget undervisningsämne i läroplanen och flera andra hade integrerat programmering i andra datateknik-kurser. Några länder, däribland Finland, har gått in för att integrera programmeringen i andra ämnen, främst i matematiken. Värt att anmärka här är att Finland hörde till den lilla mängden länder, fem stycken, som inte hade programmering som varken obligatoriskt eller valfritt i gymnasiestudier på nationell nivå. Finland är enda landet som planerade att ha programmering enbart som ett ämne integrerat med andra ämnen.

## Fördelar och nackdelar med programmering i skolan

Enligt James T. Fey (1989) handlade den första användingen av datorer i undervisning av matematik praktiskt taget alltid om programmering, men programmeringen minskade markant under 1980-talet då man inte kunde påvisa att programmering skulle ha medfört stora förbättringar i elevernas matematiska kunnande. Detta innebar också att forskning och diskussion kring ämnet avtog.

Enligt Blume & Schoen (1989) kunde man förvänta sig att elever som kunde programmera också skulle vara mera systematiska i både planerings- och utförandeskedet, utföra mera successiva approximationer, använda sig mer av variabler och ekvationer samt att sannolikare kontrollera och korrigera eventuella fel i sina lösningar. Däremot visade det sig att trots den logiska kopplingen mellan programmering och matematik kunde man inte påvisa starka bevis på skillnad mellan programmerare och icke-programmerare i fråga om problemlösning.

James T. Fey (1989) menar däremot att det är svårt att påvisa skillnader i elevernas prestationer angående problemlösning oberoende av vilken pedagogisk metod som används, och anser därför att det är för tidigt att ge upp. Fey påpekar att det kan finnas andra fördelar med programmering som inte märkts när man försökt hitta dramatiska effekter i svåra undervisnings- och inlärningsuppgifter som problemlösning. Fey menar vidare att det är möjligt att vi inte ännu känner till de mest effektiva sätten att undervisa programmering, eller ännu mindre kunna hjälpa eleverna att inse kopplingen mellan programmering och matematik.

Enligt Broley och Saint-Aubin (2018) har ironiskt nog den teknologiska utvecklingen varit orsaken till både nedgången och återupplivningen av programmering i undervisning av matematik. I dagens digitala era där våra elektroniska apparater kan skissa upp en graf för en funktion genom att i bästa fall endast yttra orden är det bra att påminna oss om tiden då detta krävde programmering för att kunna utföras. Dessa verktyg gjorde programmering till en mer eller mindre onödig kunskap, men har nu igen blivit nödvändigt för att förstå hur dessa ”svarta lådor” fungerar.

Broley och Saint-Aubin (2018) tar också upp en möjlig orsak till varför tidigare forskning inte påvisat någon större nytta av programmering: i de flesta forskningarna definieras inte vad forskarna avser med programmering, och där man utfört definieringen råder ingen konsensus om vad termen skall betyda. I bland kan till och med olika programmeringsspråk delas in i olika kategorier, där språk som Fortran, Java och C++ är programmering, medan Maple, R och MATLAB inte är det.

Broley och Saint-Aubin (2018) identifierade sex nivåer av en individs interaktion med programmering:
L0: kan endast observera resultaten av ett program
L1: kan manipulera gränssnittet av ett program
L2: kan läsa och förstå programkoden
L3: kan modifiera koden för att åstadkomma något nytt
L4: kan konstruera programkod, då vissa element färdigt givna
L5: kan skapa ett program, inkluderande utveckling av en algoritm och programmering

Dessa nivåer skall ses som intervall som kan överlappa varandra i ett kontinuerligt spektrum, beroende på hur självständigt en elev kan utföra uppgifterna.

Utgående från dessa definierade nivåer torde man kunna göra några antaganden för skolssystemet i Finland, dock finns inga exakta givna riktlinjer för detta. Grundskolans lägre klasser rör sig bland nivåerna L0 – L3, grundskolans högre klasser borde ha som mål nivåerna L2 – L4 och för gymnasiets del kan man tänka sig L3 – L5. Naturligtvis kan grundskolan nå upp till nivå L5 för elever som valt programmering som valfri tillvalskurs. Dessa nivåer kan ses som en variation av Blooms taxanomi som kanske lämpar sig bättre för programmering i skolan än den egentliga Blooms taxanomin, vilken hanteras senare i texten.

Jacobsson och Melander (2018) har sitt examensarbete intervjuat lärare i matematik i Sverige med varierande kompetensnivå i programmering om deras åsikter kring programmering som en del av undervisningen i matematik på gymnasienivå. Alla intervjuade lärare nämnde tidsaspekten som en viktig faktor och menade att de inte visste hur de skulle få tiden att räcka till när matematikkursernavar fullspäckade redan från tidigare. Lika som i Finland har man inte heller i Sverige bestämt exakt vad som skall höra till programmeringen och detta förundrade lärarna.

## Koppling mellan matematik och programmering

Kivelä (2004), lektor i matematik vid Tekniska Högskolan i Helsingfors, har i ett dokument skrivet år 2004 tagit upp synpunkter som än i dag är valida angående användning av dator i matematiken. I dokumentet tar han i flera olika exempel upp problematiken kring tudelningen, hur mycket skall studeranden förstå matematiken och hur mycket kan man få direkt uträknat av datorn. Han tar bland annat upp ett exempel, som antagligen härstammar från 1600-talet av Simon Kexlerus, Finlands första professor i matematik: ”Du har olika viner, som kostar 3, 5, 8 respektive 10 mark per flaska. Ta tio fulla flaskor vin och gör en blandning av dessa som kostar 6 mark per flaska. Hur många flaskor av varje vinsort skall du ta?”.

Kivelä (2004) menar att man kunde lösa problemet enkelt genom att göra ett kort program och pröva alla möjligheterna, eller så kunde problemet lösas mera matematiskt

med Diofantiska ekvationer. Han avslutar problemet med frågan ”Det vore intressant att veta hur Kexlerus själv löste problemet?”.

Grandell m.fl. (2006) menar att kunskaper i programmering kan vara till nytta då man skall kommunicera med personer som studerat datavetenskap. De hänvisar också till tidigare studier som påvisar kopplingen mellan programmering och problemlösning.

Vid Linköpings universitet i Sverige har man försökt öka kopplingen mellan matematik och programmering för nya studerande på programmen för datateknik och mjukvaruteknik. Att påvisa kopplingen från matematik till programmering är enkelt med begrepp som funktioner, variabler och rekursion. Däremot är kopplingen åt andra hållet, alltså kopplingen från programmering till matematik, betydligt mycket svårare att påvisa. De ger studerandena ett Python program som verktyg att använda vid lösning av inlämningsuppgifter. Som exempel nämner de en funktion som är rekursivt definierad och som ett annat exempel inom kombinatorik där studerandena kan kontrollera sina svar genom att procedurellt räkna ut det med programmering. Heintz, Färnqvist och Thorén (Heintz 2015) undersökte därefter med en enkät studerandenas tankar kring matematik och programmering. Överlag ser alla nyttan i att kunna matematik för att kunna programmera. Däremot när man frågar huruvida studerandena kan använda eller använder programmering som metod när de skall lösa matematiska uppgifter blir resultatet betydligt lägre.

## Programmering är svårt

Tyvärr finns det inte mycket forskning kring studerande i gymnasieålder eftersom focus i främsta hand verkar ha varit för barn i grundskoleålder. Gomes & Mendes (2007) har forskat i frågan om varför programmering är svårt för ungdomar som påbörjar universitetsstudier, vilket betyder att dessa objekt är ungefärligen i samma ålder som studerande i gymnasiets sista år. Gomes och Mendes (2007) tar fram fem olika punkter där svårigheterna eller problemen kan finnas:

- Undervisningsmetoderna
    - undervisningen är inte tillräckligt personlig
    - lärarens strategier motsvarar inte alltid studerandens inlärnings stil
- Studiemetoderna
    - studeranden använder fel studiemetod

    - studeranden arbetar inte tillräckligt för att uppnå kompetens i programmering
- Studerandens förmåga och attityd
    - studeranden har inte tillräckliga logiska eller matematiska kunskaper
    - studeranden saknar specifik programmeringskunskap
- Programmeringens natur
    - programmering kräver en hög nivå av abstraktion
    - programmeringsspråk har en komplex syntax
- Psykologiska effekter
    - studeranden är inte tillräckligt motiverade

Enligt egen erfarenhet som lärare i datateknik, inkluderande programmering, på högstadiet kan jag ganska långt hålla med ovanstående lista på svårigheter som eleverna har i programmering. Eftersom programmering är något helt nytt för eleverna borde det finnas mera tid för personlig handledning då svårigheter uppstår, men detta kan vara svårt att verkställa då gruppstorleken kan vara stor och man är enda läraren i undervisningssituationen. Därtill kan man ganska ofta skönja det faktum att eleverna nog skulle vilja kunna programmera, men att de inte är villiga nog att sätta den tid på övningar som skulle krävas. Programmeringens abstraktion har inte varit ett så stort problem i de kurser jag haft i programmering, om sakerna förklaras noggrant och detaljerat har eleverna ganska snabbt insett hur t.ex. olika datastrukturer fungerar. Programmeringsspråkets syntax är naturligvis alltid ett hinder i början, men lättar efter tillräckliga övningar i att skriva programkod. Den sista punkten som Gomes och Mendes tar upp, motivationen, är en sak som säkert är av betydande roll, men eftersom de programmeringskurser jag haft i högstadiet varit tillvalskurser har motivationen bland eleverna ofta varit relativt hög. I fråga om programmering som en obligatorisk del av undervisningen för alla elever kan situationen säkerligen vara en annan. Själv tror jag att om man väljer lämpliga programmeringsuppgifter kan man också höja elevernas motivationsnivå, t.ex. spelprogrammering brukar intressera de flesta.

Gomes och Mendes (2007) ger förslag på hur man kan motarbeta dessa problem i undervisningen, men dessa gäller främst för universitetsstuderanden och kan möjligtvis vara svåra att tillämpa på studeranden på gymnasienivå. Dock påvisar listan på de fem principiella problemen att programmering inte är ett lätt ämne för studeranden.

## Matematikläraren som lärare i programmering

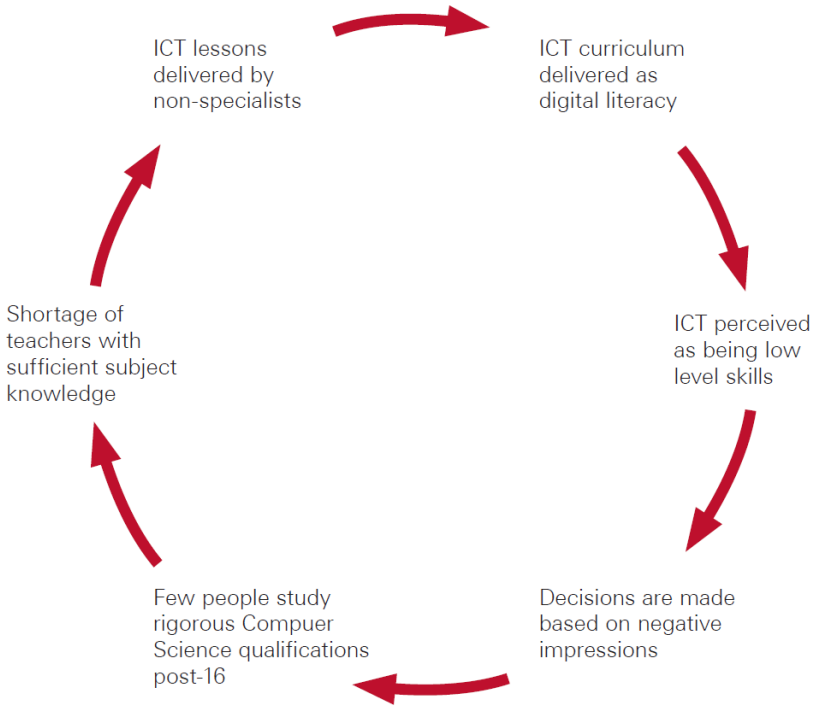
Programmering har inte varit en del av undervisningen i matematik förrän i och med läroplanen för den grundläggande undervisningen i Finland av år 2014 och först är på kommande till gymnasieskolorna i Finland år 2021. Detta betyder att en stor del matematiklärare inte studerat programmering under sin studietid vid universitet och möjligtvis inte heller under studier på annat håll.

Suominen (2019) har i sin pro gradu avhandling beskrivit hur en kurs vid namn ”Ohjelmointi matemaatikan opetuksessa” genomfördes i Helsingfors Universitet våren 2017 då kursens innehåll reformerades. Före denna kurs hade man under två år tidigare anordnat en motsvarande kurs för programmering i matematikundervisningen, men därförinnan har inga motsvarande kurser ordnats vid Helsingfors Universitet. Naturligtvis har matematiklärarstuderanden kunnat delta i fakultetens programmeringskurser på datavetenskapslinjen.

Kursen där Suominen var med som hjälphandledare och även skrivit sin avhandling om kan vara bra för kommande lärare som inte har tidigare kunskaper i programmering, men även längre komna inom programmering kan ha nytta av kursen då den inte enbart handlar om programmering. Programmeringsdelarna i kursen gjordes i både Scratch och Python. Medan programmeringsuppgifterna gjordes funderade man en aning på vilka typer av programmeringsuppgifter som kunde vara lämpliga i skolvärlden. Därtill bildade man bland de studerande studiecirklar där man skall söka fram vetenskapliga artiklar kring programmering i skolan.

Undertecknad var själv en av de nämnda sju studeranden som tog del av kursen. Även om kursen främst var inriktad på programmering i grundskolan deltog jag i ett större grupparbete där vi utformade en plan för undervisning av programmering angående vektorer för gymnasiestuderanden med Python som programmeringsspråk och ”turtle”-grafik. Detta övningsarbete finns öppet tillgängligt på internet för dem som vill använda det, arbetet består av handledning för både lärare och studerande.

The Royal Society (2012) i Storbritanien gjorde en studie 2012 angående hur läroplanen i England kunde formas för framtiden innehållandes programmering. De lyfter fram flera problem vilka leder till en ond cirkel som måste brytas. Själva läroplanen samt lärarnas kunnande är de saker som kan förbättras på statlig nivå.




Figur 1. Ond cirkel med icke-specialiserade lärare som en del i cirkeln.

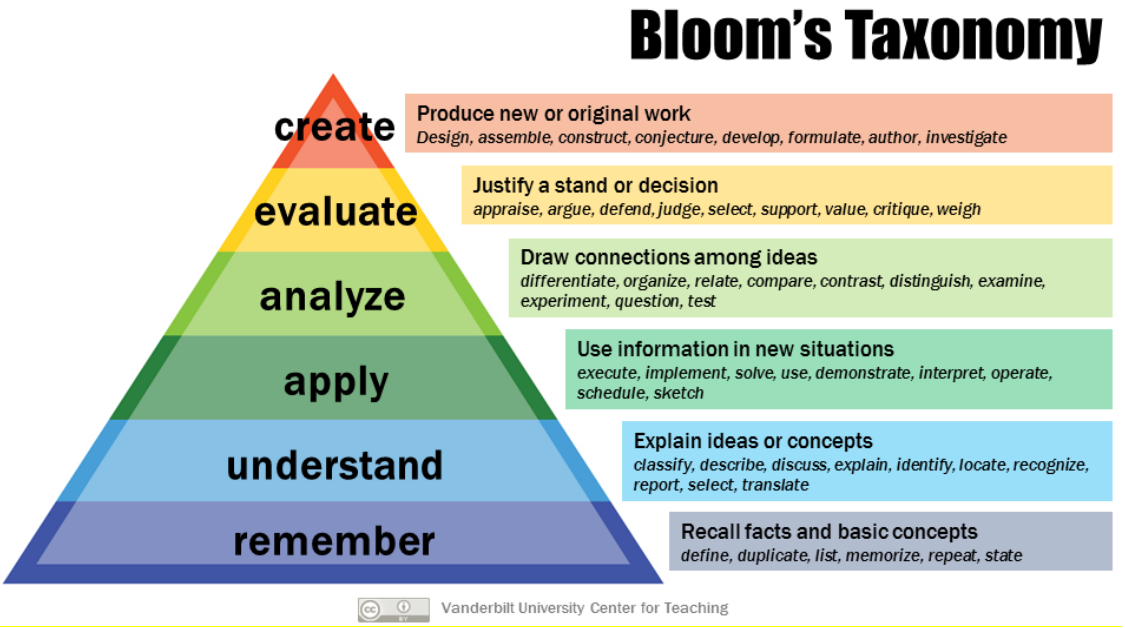
Denna möjliga brist på kunskap inom programmering bland matematiklärare kommer att vara ett större problem i grundskolan än i gymnasiet eftersom gymnasiet har programmering endast i en kurs i långa matematiken. Men även om problemet är mindre i gymnasiet kan inte detta faktum förbises och alla matematiklärare i gymnasiet bör fortbilda sig till även denna kunskap. Och som redan ovan nämnts är programmering inte en enkel sak att bemästra, därför kommer fortbildningen att vara krävande.

# Hur bedöma programmering i matematiken

Programmering kan vara ytterst tidskrävande då redan planerandet av algoritmen och överförandet av den till ett programmeringsspråk kan vara krävande kommer därtill kommer debuggningen av programkoden. De enklaste felen, syntaxfelen, är ofta ganska enkla att hitta, men logiska fel i koden kan vara betydligt svårare att märka. Man kan lära sig vissa debuggningsstrategier vilka hjälper till, men inlärning av dessa tar också sin tid. Här stöter man på svårigheten med hur tiden skall disponeras mellan den egentliga matematiken och programmering. Utgångsläget, då det undervisade ämnet är matematik, bör vara att matematiken kommer i första hand och programmeringen först därefter. Detta problemmoment uppstår oberoende av till vilket ämne programmeringen än sammanknyts med i skolan, enda lösningen vore att programmering utgjorde en del av ett eget undervisningsämne.
Om programmeringen upptar endast en liten del av lektionstiden och tiden för hemuppgifter kan man tänka sig att goda kunskaper i programmering skulle kunna bidra till ett bättre vitsord, men däremot skulle sämre kunskaper inte sänka vitsordet. Om tidsfördelningen är en annan måste också programmeringskunskaperna beaktas i vitsordet som både en höjande eller sänkande faktor.

Men hur skall programmeringskunskaperna då bedömas? Man kan gärna inte i en matematikkurs ha prov angående programmering, och att tekniskt genomföra möjligheten att ha programmeringsuppgifter i ett prov kan vara mycket svårt, speciellt då gymnasierna i dag använder sig av Abitti-systemet som inte tillåter uppkoppling till Internet. Utan uppkoppling till Internet måste dokumentationen till det aktuellta programmeringsspråket laddas upp som en PDF-fil, dock är den officiella dokumentationen för de flesta programmeringsspråk svårläst för nybörjare. För nybörjare är det lättare att kunna söka på Internet efter exempel på hur en funktion skall användas och hurdan dess syntax är. Detta kunde delvis åtgärdas genom att inkludera i Abitti-distributionen en bra kodeditor, men då återkommer vi till problematiken kring val av programmeringsspråk, vilket hanteras närmare i nästa kapitel.

Om problemet ses ur ett programmeringsperspektiv kunde man tänka sig att applicera Blooms taxonomi på problemet, nedan en bild som anger hur den reviderade versionen av Blooms taxonomi lyder på engelska.




Figur 2. Bilden är gjord av Vanderbilt University Center for Teaching. De engelska orden nerifrån uppåt motsvaras i svenskan av verben: minnas, förstå, tillämpa, analysera, värdera och skapa.

Starr, Manaris och Stalvey (2008) har gjort detta och identifierat vilka delar av Blooms taxonomi som hör till olika nivåer av kunskap inom programmering. Nybörjare skall kunna minnas och förstå, medelgoda programmerare skall kunna tillämpa och analysera samt experter skall kunna värdera och skapa. Starr et al. ger också exempel på hurdana frågor de olika nivåerna kan tänkas innehålla (t.ex: tillämpa – ”vad är programmets utdata?”, värdera – ”vilket är bättre av två program som utför samma funktion, och varför?”).

Som läget är med den nya läroplanen, så verkar det vara upp till var och en enskild lärare att bestämma hur mycket vikt man sätter på programmeringskunskaper vid bedömning av kursen.

## Automatiska tester

Det optimala sättet att bedöma programmeringskunskaper är naturligtvis att läraren går igenom den inlämnade koden manuellt, men detta är ytterst tidskrävande då det aldrig finns endast ett korrekt sätt att skriva en kod som skall utföra en bestämd uppgift. Om gruppstorleken är liten och programmeringsuppgifterna de studerande skall utföra är

simpla kan detta fungera väl. Däremot om uppgifterna är mera avancerade och elvantalet eller antalet uppgifter är stort kan arbetsmängden bli för stor för lärarens del.

Därför är lösningen automatiserade system som granskar att koden fungerar enligt anvisningarna. Dylika automatiska system finns på marknaden, och de används även i undervisning. Helsingfors Universitet använder i utvecklarverktyget Netbeans modulen TestMyCode (TMC) som utför dessa kontroller. TMC kan användas med flera olika programmeringsspråk, men eftersom Netbeans primärt är ett verktyg för programmering i Java så fungerar även TMC bäst för Java.

Hellas (2017) som varit en av de drivande personerna när Helsingfors Universitet har skapat s.k. MOOC-kurser (eng. Massive Open Online Course) i programmering hanterar i sin doktoravhandling också hur TMC-systemet fungerar i MOOC-kurserna. Några av de viktigaste aspekterna i TMC för grundskolan och gymnasierna är följande:

- dubbelriktad feedback angående uppgifterna mellan studerande och lärare
- samlar data om studerandens fortskridning i processen för senare analys och kontroll av plagiat

I fall det valda programmeringsspråket i gymnasiet är Java är Netbeans med TMC ett alternativ. Dock bör här påpekas att dessa kontroller inte sker helt automatiskt utan i kontrolleringsmodulen måste olika tester skapas som sedan en i gången körs gentemot koden och i fall alla tester avklaras godkänt tolkas uppgiften som korrekt utförd.

Dessa tester som bland annat TMC körs är så kallade enhetstester (eng. ”unittest”), och dessa kan tillämpas i de flesta programmeringsspråken. Att skapa dessa tester kan vara tidskrävande, och därför vore det bra om det fanns färdiga programmeringsuppgifter i läromedlen och till dem hörande tester. I det fallet att läraren förutom att planera programmeringsuppgifter även måste göra enhetstesterna samt i värsta fall även sätta upp en server för testningssyfte är vi långt ifrån det en matematiklärare egentligen skall syssla med.

Ihantola (2011) går utförligt igenom i sin doktorsavhandling olika verktyg och metoder för hur automatiska kontroller kan genomföras. I slutet av avhandlingen går han in

djupare in på hur ett dylikt system kan byggas upp, detta upplyser de tekniska svårigheterna med skapande av automatiska system för kontrollering av programkod.

Problemet med automatiserade kontroller av programkoden oftast kräver att utdatat är exakt lika som det är specificerat i testerna. Detta innebär inte enbart problem med textsträngar som kan innehålla extra mellanslag och därför inte godkänns, men också då man skall lösa numeriskt någonting och man rör sig vid gränserna för noggrannheten programmet räknar med. Ett typiskt exempel för de flesta programmeringsspråk är att 0,1 + 0,2 = 0,30000000000000004 vilket inte är exakt samma som det korrekta svaret 0,3 (denna inexakthet beror på hur flyttal fungerar i en dator, mera om detta i kapitlet om flyttal).

# Val av programmeringsspråk

Val av programmeringsspråk är inte en enkel uppgift, då det finns så många olika språk att välja mellan. Dock borde väl här i främsta hand tas i beaktande språkets svårighetsgrad och hur lämpat det är som ett första programmeringsspråk för studerandena. I andra hand måste också språkets tillgänglighet och andvändningsgrad beaktas.

Under 1970- och 80- talet då programmering började komma till skolorna runt om i världen var valet relativt enkelt då det fanns språk som var planerade för nybörjare, här kan nämnas två av de mest kända: Basic och Pascal. I Basic utgick man från att det skall vara så enkelt som möjligt att komma igång med programmeringen och meningen var aldrig heller att språket skulle användas professionellt. Pascal hade mer eller mindre samma utgångsläge, men man ansåg att Pascal också skulle lämpa sig för professionellt bruk, även om det fanns vissa restriktioner som kunde vålla problem, därför utkom det senare en ny förbättrad version Modula-2, som också understödde objektorienterad programmering.

Edsger Dijkstra (1982) som var en av de mest framstående personerna inom datavetenskapen under senare halvan av 1900-talet hade starka åsikter om Basic som programmeringsspråk, han skrev bland annat följande ”Det är i praktiken omöjligt att undervisa god programmering till studenter som tidigare har varit utsatta för Basic: som programmerare är de mentalt stympade utom allt hopp om regeneration”. Däremot bör man märka att Dijkstra som var en stark förespråkare för programmeringsspråket ALGOL60 hade som vana att kritisera de flesta andra programmeringsspråk i motsvarande ordalag, även FORTRAN, COBOL, PL/1 och APL fick hård kritik i samma kapitel där han skrev ovanstående om Basic. Av dessa är det däremot endast uttrycket om Basic som fortlever som ett bevingat ordspråk, och kanske därför också en bidragande orsak till att Basic inte lägre används i nämnvärd utsträckning. I detta sammanhang kan man också notera vad Dijkstra sade om programmering och matematik: ”Programmering är en av de svåraste branscherna inom tillämpad matematik; de sämre matematikerna bör förbli rena matematiker”.

I Sverige gjorde man ett beslut i början av 1980-talet att alla skolor skulle använda Esselte Compis-datorn (Halén 2001), och därmed blev det programmeringsspråket Comal som fanns färdigt installerat på datorn som blev språket som användes. Comal var en mera strukturerad variant av Basic utvecklad i Danmark. I flera svenskspråkiga skolor i Finland användes då också samma Comal som första programmeringsspråk.

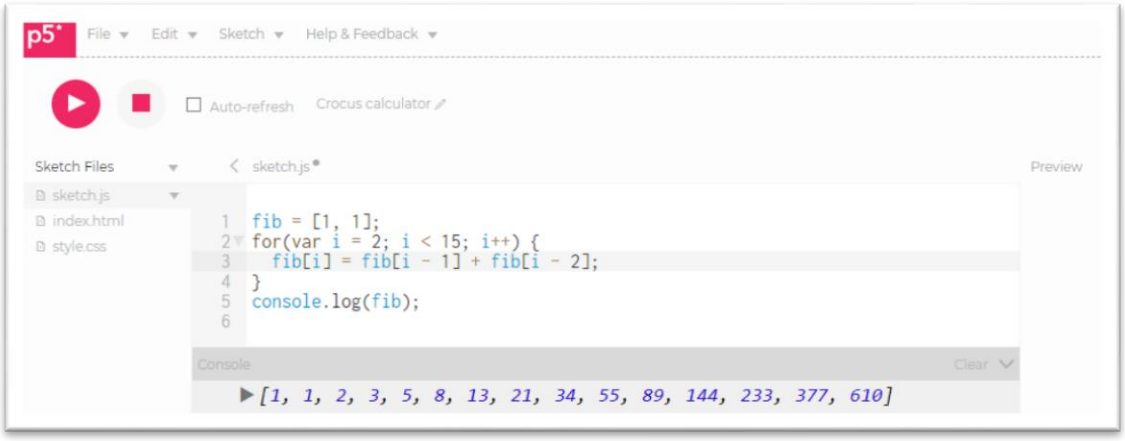
Men vilka alternativ till programmeringsspråk kan vi då välja mellan i dagens läge? Det finns inga bra alternativ till enkla språk i dag då Basic och Pascal med sina olika modernare varianter har mer eller mindre försvunnit och inga nya ersättande språk har tillkommit. En detalj som man kan räkna som en del av programmeringsspråkets enkelhet är om syntaxen som språket använder i stort är lika som för andra programmeringsspråk. Efter att C (och senare C++) blivit ett mycket använt och inflytelserikt språk har flera andra språk därefter till en viss mån kopierat samma syntax som C. Flera kända stora programmeringsspråk som används idag som använder sig av en C-liknande syntax är förutom C och C++ bland annat Java, C#, PHP och JavaScript, vilka alla hör till de mest använda programmeringsspråken enligt programmerargemenskapen GitHub (2019).
Förutom C-syntax, kunde man möjligtvis välja programmeringsspråk som hör till Lisp-familjen, eller Python som snabbt har blivit ett populärt språk bland matematiker. Däremot bör man antagligen förbise språk som R, MatLab o.s.v. även om dessa är nog så användbara språk inom olika grenar av matematik (R för statisikrelaterade uppgifter och MatLab för matriser), då dessa har en syntax som kan anses vara ganska annorlunda än andra mera allmännyttiga programmeringsspråk.
I denna text faller valet för programmeringsspråk på JavaScript, inte av den orsaken att det skulle vara det bästa programmeringsspråket för matematiska uppgifter, hade detta varit det viktigaste kriteriet hade valet säkerligen fallit på Python. JavaScript har flera nackdelar, men det har också vissa oöverträffbara fördelar, nämligen JavaScript kan köras på alla datorer som har en webbläsare installerad och därtill är JavaScript ett av de mest använda språken i dagens läge och studerandena kan då använda sig av de lärda kunskaperna i JavaScript till mycket annat också då användningen är så utbredd.

## Att använda JavaScript

Som tidigare nämnts kan alla webbläsare köra JavaScript kod och detta har också lett till att JavaScript är ett av de mest använda språken i dag (GitHub 2019). Men alla

program är inte optimala för att köras i en webbläsare och därför kan man också köra JavaScript som vilket annat program som helst genom att installera Node.js. Förmodligen är det lättare för nybörjare att köra koden i webbläsaren då det i moderna webbläsare finns effektiva debuggningsverktyg inbyggda för JavaScript.

Om JavaScript körs i en webbläsare måste man inse att en webbläsares uppgift är att visa en webbsida och därför måste JavaScript-koden byggas in i en enkel webbsida före den kan köras. En annan möjlighet är att man använde en onlinetjänst där allt detta sker i bakgrunden, exempelvis ”p5.js Web Editor” (https://editor.p5js.org/), där användaren vid behov också enkelt kan se vilka filer som behövs för att bilda en webbsida.




Figur 3. Exempel på p5.js Web Editor med ett kort program som skriver ut Fibonacci-tal.

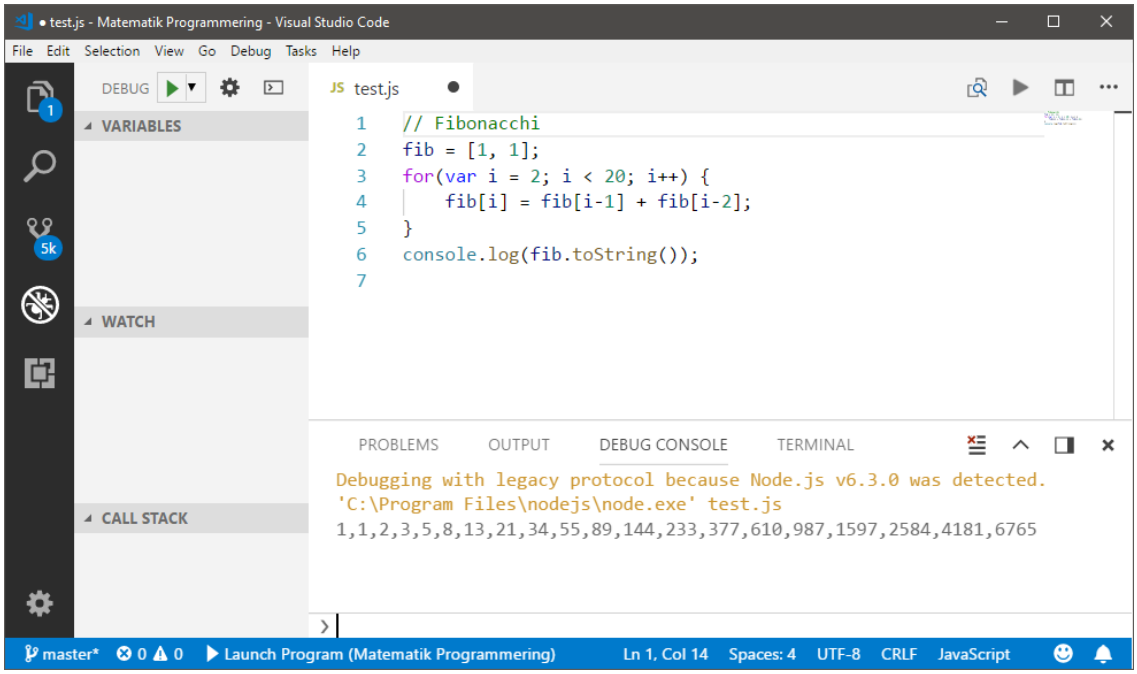
Som exemplet med p5 Web Editor visar finns det alltid en index.html fil i filstrukturen som bildar den nödvändiga hemsidan där JavaScript sedan körs. Man kan naturligtvis göra denna index.html fil själv också (namnet på filen får vara annat än index, men filnamnstillägget bör vara .html eller .htm), här är den kortaste och simplaste HTML-filen som behövs:

```
<!DOCTYPE html>
<head>
    <title>Document</title>
</head>
<body>
<script src="javascript.js"></script>
</body>
</html>
```

Denna kod skapar en sida som följer HTML5-standarden och läser in en JavaScript fil vilken körs automatiskt, vill man koppla in flera JavaScript filer till sidan skapar man

flera kodrader motsvarande den med HTML-taggarna $< script >$ och $</script >$. JavaScript kan också skrivas direkt in i HTML-filen, men detta är inte att föredra då koden kan bli svårläst då flera olika symboliska språk blandas ihop i en fil.

JavaScript är ett skriptspråk som inte kompileras till en körbar fil, vilket gör att utvecklingen och debuggningen av koden är snabbt och smidigt. Om man föredrar en riktig kodeditor för att skriva koden, så finns det flera att välja mellan, en av de mest använda i dagens läge är Microsoft Visual Studio Code, som är ett projekt med öppen källkod och därför gratis för alla användare och fungerar också på alla moderna plattformer. Med Node.js installerat kan JavaScript program köras i Visual Studio Code utan HTML-delen. Därtill är Visual Studio Code också mycket användbart som editor för de flesta andra programmeringsspråken eftersom programmet är modulärt och man enkelt kan installera nya komponenter till programmet. Som kuriositet kan nämnas att Visual Studio Code i sig är ett JavaScript program som använder ramverket Electron för att skapa ett fristående program.




Figur 4. Exempel på Visual Studio Code och ett program som skriver ut Fibonacci-tal i den inbyggda konsolen.

## Problem med JavaScript

Vissa svårigheter kan uppstå med syntaxen i JavaScript, men då den är baserad på C-syntaxen uppstår motsvarande svårigheter i de flesta programmeringsspråk. Lisp-varianter och Python använder inte samma syntax och därför definieras inte block som strukturer i dessa språk med problematiska klamrar, men däremot är saken inte enklare med den stora mängden parenteser inne i varandra som krävs även i enkla Lisp program, eller indenteringen (indrag) med mellanslag eller tabulatorsteg i Python då det kan bli svårt bryta ett block på rätt ställe om det finns flera nivåer av indentering.

Ett annat problem som finns i JavaScript men inte i de flesta andra programmeringsspråken är att JavaScript använder sig av dynamisk typ för variabler, vilket betyder att en variabel kan vid ett skede innehålla ett numeriskt värde och senare exempelvis en teckensträng. Variabler med dynamisk typ är enklare för nybörjare att använda då man inte behöver fundera över vilken typ variabeln skall vara, men i större projekt kan detta leda till kod som inte alltid fungerar korrekt. Orsaken till detta är att språk som har variabler med strikta typer (typen kan inte ändras efter den första deklarationen av variabeln) och då kan editorn eller kompilatorn granska variabeltyperna redan före koden skall köras eller kompileras. TypeScript är en variant av JavaScript som följer alla definitioner för JavaScript, men har därtill en del andra egenskaper, bland annat strikta typer för variabler. TypeScript måste kompileras till standard JavaScript före koden kan köras.

# Fundamentala matematiska begrepp inom programmering

Även om kursen i fråga bär namnet Algoritmer och talteori, skall jag här ändå också gå igenom grunderna i det logiska och matematiska begrepp som behövs för att förstå hur en dator fungerar och därmed även hur dessa används inom programmering. Innehållet i detta kapitel behöver inte studerande kunna i djupet för att kunna lösa problem med programmering, men grunderna är dock nödvändiga. De logiska begreppen och funktionerna behövs för att kunna styra kondinationaliteter i programmet, bland annat ”if”-, ”for”- och ”while”-satser. De matematiska begreppen behövs ifall man vill optimera koden och även för att förstå varför datorn i bland räknar fel då modellen för representation av numeriska värden inte har tillräckligt hög upplösning.

## Binära talsystemet

Binära talsystemet har basen 2 och därmed endast 2 siffror, 0 och 1. Även om det binära talsystemet inte behövs speciellt ofta i högre nivå av programmeringsspråk, som i detta fall JavaScript, så har det sina fördelar att förstå grunderna i det då själva processorn i en dator egentligen endast kan hantera binära tal. Förståelsen av det binära talsystemet och hur en processor är uppbyggd hjälper till att förstå varför vissa numeriska värden som annars kan verka mystiska dyker upp så ofta inom programmering.

Det binära talsystemet fungerar exakt lika som det vardagliga decimaltalssystemet, med sifferpositioner och siffror i dessa positioner, som i det binära talsystemet oftast kallas för bits (förkortning från engelskans ”binary digit”). Här hanteras endast positiva heltal även om det naturligtvis är möjligt att hantera alla reella tal i binär form. Vi kan bestämma korrelationen mellan antal siffror i 10-talssystemet och bitar enligt följande: antal bitar som behövs för att kunna representera 10 olika värden (siffrornas antal i 10-talssystemet) kan bestämmas enligt $2^x = 10 \Rightarrow x = \ln(10) / \ln(2) \approx 3{,}322$. Bitarna i ett binärt tal indexeras med 0 för biten längst till höger.

I datavetenskapliga sammanhang brukar bitarnas antal ofta vara delbara med 8 och de mest använda antalen är 8, 16, 32 och 64, undantag finns dock. Denna delbarhet med 8 kommer främst från processorernas arkitektur vilket är ett ämnesområde som förbises här. Bitarna indexeras med index 0 för biten längst ut till höger, detta är biten med den

minsta betydelsen för talets värde och kallas ofta också för LSB (eng. least significant bit).

Tabell 1. De binära sifferpositionernas motsvarande värden i tiotalssystemet.

| $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ | $b_1$ | $b_0$ |
|---|---|---|---|---|---|---|---|
| $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

Som det framgår ur tabellen kan positionens värde för varje bit räknas ut enligt följande: då $i$ är bitens position är $b_i = 2^i$.

## Addition av binära tal

Additionen för binära tal fungerar exakt lika som med tal i 10-talssystemet. Ser vi på enskilda bitar är de enda möjligheterna för addition följande: 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1 och 1 + 1 = 10. Vid addition av bitar kan summan bestå av två bitar, de extra biten är inget annat än en minnessiffra.

Tabell 2. Sanningsvärdestabell för binär addition med minnessiffra in och ut.

| $A$ | $B$ | $C_{in}$ | $C_{out}$ | $Sum$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

Ovanstående tabell kan överföras till logiska konnektiver och därmed också till logiska grindar som en stor del av dagens digital elektronik baserar sig på. Dessa logiska grindar skulle bilda en helhet som kallas för en 1-bits heladderare eftersom den också hanterar en inkommande minnesbit. Om man börjar sålunda att första minnesbiten har värdet 0 kan man kedja ihop flera av dessa adderare där den föregåendes utgående

minnesbit blir del följandes inkommande, på detta sätt kan vi enkelt göra en krets som utför addition på ett större binärt tal. Det bör dock påpekas att även om denna konstruktion är möjlig, så är den långt ifrån effektiv då minnesbiten måste droppa vidare i kedjan och varje addition tar minst en cykel av kretsens klockfrekvens. Överföringen av denna sanningsvärdestabell till logiska konnektiver tas upp i kapitlet om disjunktiv normal form.

## Logik

Här går vi kort igenom hur de vanliga logiska konnektiverna fungerar och hur dessa även kan användas i större skala med hjälp av VENN-diagram, DeMorgans-lagar och Disjunktiv Normal Form. Även om man inom matematiken allmänt använder beteckningarna SANN och FALSK som sanningsvärden för de logiska konnektiverna använder jag här de Booleska varianterna 1 respektive 0, då dessa har samma betydelse men också går att använda för de enskilda siffrorna i ett binärt tal.

### Konnektiver

Konnektiven inom satslogiken som används i matematik är negation, konjunktion, disjuktion, implikation och ekvivalens, men i fråga om programmering är det i första hand endast de tre förstnämnda som används. Som vi kommer att märka kan alla konnektiver enkelt beskrivas med de tre som används i programmering. Predikatlogikens kvantifikatorer används inte i programmering, och måste oftast hanteras med hjälp av loopar.

Tabell 3. Tabell över logiska konnektiver och deras motsvarigheter i JavaScript.

| ***Symbol*** | ***Namn*** | ***Betydelse*** | ***JavaScript*** |
|---|---|---|---|
| ¬ | Negation | Inte | ! |
| ∧ | Konjunktion | Och | && |
| ∨ | Disjunktion | Eller | \|\| |
| → | Implikation | Om.., då… | (används inte) |
| ↔ | Ekvivalens | Om och endast om | (används inte) |

I fråga om design av logiska kretsar stöter man ofta på ett alternativt sätt att markera konnektiven: negation med ett streck över en proposition eller sats, konjunktion

betecknat som multiplikation och disjunktion betecknat som addition (Wiatrowski 1988). Dessa noteringssätt är inte att föredra i gymnasiematematiken då det är bättre att studeranden lär sig ett sätt att beteckna logiska satser väl än i värsta fall dåligt två olika sätt. Därtill hör inte denna typ av notering till standardnoteringen i matematik.

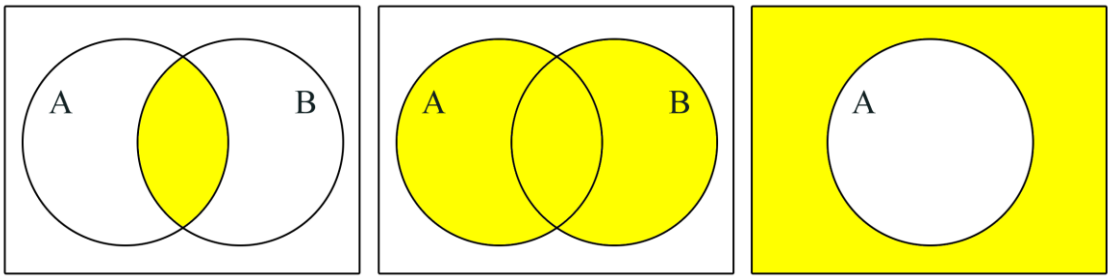
Tabell 4. Sanningsvärdestabeller för de logiska konnektiverna.

| $A$ | $B$ | $\neg A$ | $\neg B$ | $A \wedge B$ | $A \vee B$ | $A \rightarrow B$ | $A \leftrightarrow B$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |

Som framgår ur sanningsvärdestabellen ovan ändrar negationen på värdet från det ena möjliga till det andra. Konjunktionen mellan två satser blir sann endast om båda satserna är sanna. Disjunktionen blir falsk endast om båda satserna är falska. Implikation och ekvivalens hanteras i stycket angående disjunktiv normal form.

### VENN-diagram

VENN-diagram kan med fördel användas för att illustrera de logiska operationerna grafiskt då propositionernas antal är litet. Diagrammen används inom logiken på samma sätt som i mängdläran. Idén är att ha ett område eller universum som innehåller en eller flera mängder som kan överlappa varandra helt fritt. Mängdlärans union motsvaras av disjunktion, snitt av konjunktion och komplement av negation.




Figur 5. Bilder på VENN-diagram.

I bilderna ovan är mängderna $A$ och $B$ ritade som cirklar. Den gula färgen representerar då $A \wedge B$, $A \vee B$ respektive $\neg A$ är sanna. $A \wedge B$ är sant då både $A$ och $B$ är sanna samtidigt, $A \vee B$ är sant då någondera eller både $A$ och $B$ är sanna, och $\neg A$ är sant då $A$ är falskt.

### DeMorgans lagar

DeMorgans lagar kan ibland vara till nytta inom programmering, men sällan för nybörjare. Däremot är det bra att kunna dessa konverteringsformler allmänt inom logiken, inom datavetenskapen hör de mera till design av logiska kretsar än programmering.

Tabell 5. Sanningsvärdestabeller som underlag för DeMorgans lagar.

| $\boldsymbol{A}$ | $\boldsymbol{B}$ | $\neg(\boldsymbol{A} \wedge \boldsymbol{B})$ | $\neg\boldsymbol{A} \vee \neg\boldsymbol{B}$ | $\neg(\boldsymbol{A} \vee \boldsymbol{B})$ | $\neg\boldsymbol{A} \wedge \neg\boldsymbol{B}$ |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |

Som man kan se ur ovanstående tabell har vi parvis uttryck som antar exakt samma värden och är därmed ekvivalenta.

Tabell 6. DeMorgans lagar.

| | |
|---|---|
| $\neg(A \wedge B) \leftrightarrow \neg A \vee \neg B$ | $\neg(A \vee B) \leftrightarrow \neg A \wedge \neg B$ |

### Disjunktiv Normal Form (DNF)

Redan tidigare har det nämnts att vi inte behöver andra konnektiver än konjuntion, disjuktion och negation för att kunna hantera även implikation och ekvivalens. Denna metod som här tas upp kan tillämpas på alla logiska satser bestående av propositioner.

Metoden går ut över det att man granskar hur man i sanningsvärdestabellen kan bestämma alla rader som blir sanna genom en konjunktion av alla propositioner eller deras negationer. Alla dessa konjunktioner slås sedan samman med disjunktioner.

Tekniskt sett kan man här se konjunktionerna som multiplikation och disjunktionerna som addition och därmed kan metoden ses som en summa av produkter (”Sum-of-Products”).

Tabell 7. Sanningsvärdestabll med exempel på hur den Disjunktiva Normal Formen skapas.

| $\boldsymbol{A}$ | $\boldsymbol{B}$ | $\boldsymbol{A \rightarrow B}$ | $\boldsymbol{A \leftrightarrow B}$ |
|---|---|---|---|
| 0 | 0 | $1 = \neg A \wedge \neg B$ | $1 = \neg A \wedge \neg B$ |
| 0 | 1 | $1 = \neg A \wedge B$ | 0 (hanteras ej) |
| 1 | 0 | 0 (hanteras ej) | 0 (hanteras ej) |
| 1 | 1 | $1 = A \wedge B$ | $1 = A \wedge B = 1$ |

Här av får man att DNF för $A \rightarrow B$ att vara $(\neg A \wedge \neg B) \vee (\neg A \wedge B) \vee (A \wedge B)$. På motsvarande sätt får man DNF för $A \leftrightarrow B$ att vara $(\neg A \wedge \neg B) \vee (A \wedge B)$. Implikationen kan göras mycket enklare genom optimering av konnektiven eller genom att istället utföra en annan metod som ger den så kallade konjunktiva normal formen (KNF eller ”Products-of-Sums”), då motsvaras $A \rightarrow B$ av $\neg A \vee B$.

DNF och KNF går alltid att använda, men resulterar inte nödvändigtvis i de mest optimala lösningarna, dock ligger inte de effektiva Karnaugh-diagrammen (Wiatrowski 1988) inom ramarna för gymnasiematematiken.

## Bitvisa logiska operationer

De logiska konnektiven kan också överflyttas till det binära talsystemet sålunda att man utför den logiska operationen skilt för varje bit i samma position. Eftersom operationerna är mycket enkla påvisas de här endast med exempel.

Tabell 8. Exempel på bitvis negation.

| INTE | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |

Tabell 9. Exempel på bitvis konjunktion.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| OCH | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Tabell 10. Exempel på bitvis disjunktion.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| ELLER | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |

Även om JavaScript understöder bitvisa logiska operationer kan de vara en aning problematiska i och med att JavaScript inte har en skild datatyp för heltal utan alla tal sparas internt som 64-bits flyttal. JavaScript konverterar flyttalet till binär form före den bitvisa operationen och efter operationen konverteras svaret igen till flyttal, detta medför att även om bitvisa operationer i en dator borde vara ytterst snabba så är detta inte fallet i JavaScript och därför används bitvisa operationer sällan i JavaScript.

Tabell 11. Symbolerna för de bitvisa operationerna i JavaScript.

| INTE | OCH | ELLER |
|---|---|---|
| ~ | & | \| |

JavaScript har även andra bitvisa operationer (XOR och diverse olika bitförflyttnings operationer) men dessa hanteras inte här då det inte finns motsvarande logiska operationer eller konnektiver för andra sammanhang.

## Andra tal än positiva heltal

Alla exempel i denna text hanterar endast positiva heltal, men självklart klarar en dator också av att hantera negativa tal samt decimaltal. För att kunna representera andra tal än endast positiva heltal måste man använda avsevärt mera avancerade metoder. För de negativa heltalens del används så kallade 2-komplementtal och för decimaltal används så kallade flyttal. Även om dessa noteringar inte nödvändigtvis behövs tas upp i gymnasieundervinsingen, så kan de om inte annat utgöra extra innehåll för dem som lätt klarar av de enklare uppgifterna i kursen.

### Negativa heltal (2-komplement)

Då den binära representationen inte har förtecken kan den inte heller som sådan användas för att representera negativa tal. Genom att välja att en bit representerar förtecknet kan detta åstadkommas, men att enbart göra detta räcker inte till om vi vill att addition mellan dessa tal skall fungera smidigt. Med smidigt avses här att den addition med negativa tal skall fungera så lika som möjligt som med positiva tal.

Som utgångsvillkor måste de binära talen ha lika många bitar, detta är i vanliga fall det som oftast råder inom datatekniken. Ofta är dessa längder 8, 16, 32 eller 64 bitar i dagens datorer. Därtill måste man observera att metoden inte alltid ger en korrekt lösning och därför måste lösningens förtecken granskas. Om termerna har motsatta förtecken uppstår det aldrig problem, men om termerna däremot har samma tecken måste också lösningen få samma förtecken. Denna granskning av förtecken kan göras relativt enkelt som en logisk sats för MSB-biten i båda termerna samt summan. Då felet uppstår har ett så kallat överflödesfel skett, det korrekta talet kan inte representeras korrekt med det antal bitar som finns tillgängligt.

2-komplementet till ett tal framställs enklast genom att först ta talets negation (1-komplement form) och sedan addera 1 till detta, MSB-biten markerar förtecknet där 0 är plus och 1 är minus.

Tabell 12. 2-komplement av talet 15 med 8 bitar, först negation, sedan addition med 1.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 15 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| Neg. | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| +1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |

Tabell 13. 2-komplement av talet 127 med 8 bitar, först negation, sedan addition med 1.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 127 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Neg. | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| +1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Därmed kan man konstatera att -15 i denna representation är 11110001 och -127 är 10000001. Här demonstreras nu två fall där additionen med 2-komplementtal går fel då 8-bitar används, felet kan kringgås genom att utöka bitarnas antal.

Tabell 14. Addition av 15 och 127 i 2-komplement form. Märk att svaret blir negativt, vilket är fel!

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| + | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| = | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |

Svaret skulle i 2-komplement form med 8-bitar bli 10001110 vilket motsvarar ett negativt tal, vilket inte är möjligt då två positiva tal adderas.

Tabell 15. Addition av -15 och -127 i 2-komplement form. Märk att svaret blir positivt, vilket är fel!

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| + | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| = | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

Här ser vi ett fall då addition av två negativa termer skulle leda till en positiv lösning, vilket inte kan stämma.

## Decimaltal (flyttal)

Förutom heltal behövs naturligtvis även de reella talen för att en dator skall kunna användas som ett analytiskt verktyg. I frågan om heltal är det enkelt att flytta dem över till användning i en dator då det alltid finns ett bestämt antal heltal i ett begränsat intervall, exempelvis finns det 256 icke-negativa heltal i intervallet [0, 255] och detta är den största mängden heltal som kan representeras med 8 bitar. Däremot gäller inte detta för de reella talen då det inom alla intervall alltid finns oändligt många reella tal. Därmed kan man inte hantera reella tal i en dator lika som heltal genom att endast begränsa intervallet för giltiga tal enligt antal tillgängliga bitar.

Istället för egentliga reella tal i en dator använder man sig av så kallade flyttal som kan representera så väl positiva som negativa tal vars absoluta värde kan variera mellan

mycket litet och mycket stort. Flyttal är endast approximationer av reella tal, men det räcker till i de flesta fallen. Då talet som representeras har ett litet absolutvärde är flyttalsrepresentationens absoluta fel litet, i annat fall blir det absoluta felet stort. Däremot är det relativa felet alltid ganska litet för flyttalsapproximationen Det relativa felets storlek är till stor grad beroende av vilken precision man använder för flyttalsrepresentationen.

Tabell 16. Karakteristikor för några flyttalsrepresentationer (IEEE 2008).

| **Namn** | **Bitar totalt** | **Bitar förtecken** | **Bitar exponent** | **Bitar mantissa** | **Exponentens förskjutning** |
|---|---|---|---|---|---|
| Binary16, ”Half” | 16 | 1 | 5 | 10 (+1) | 15 |
| Binary32, ”Single” | 32 | 1 | 8 | 23 (+1) | 127 |
| Binary64, ”Double” | 64 | 1 | 11 | 52 (+1) | 1023 |
| Binary128, ”Quad” | 128 | 1 | 15 | 112 (+1) | 16383 |

Flyttalen representeras med det antal bitar som ovanstående tabell anger enligt formen $\pm mantissa \cdot 2^{exponent}$. Mantissans värde normaliseras till intervallet $[1, 2[$. Därmed är heltalsbiten alltid 1 och mantissan kan representeras av endast decimaldelen för att spara utrymme. För att exponenten skall kunna representera både negativa och positiva värden förutom noll adderas ett förskjutningsvärde till den egentliga exponenten, detta förskjuter den effektiva nollan enligt förskjutningsvärdet. Förskjutningsvärdet är alltid $2^{antal\ bitar\ i\ exponenten-1} - 1$, som också kan avläsas från ovanstående tabell.

Exempel på flyttals representationer med enkel (”single”) precision:

$+0{,}5 = +1{,}0 \cdot 2^{-1}$

förtecken: $(+) = 0_2$

exponent: $-1 + 127 = 126 = 0111\ 1110_2$

mantissa (decimaldel): $1{,}0 = ,000\ 0000\ 0000\ 0000\ 0000\ 0000_2$

binär form:

$0011\ 1111\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$

$-1389{,}568 \approx -1{,}357 \cdot 2^{10}$

förtecken: $(-) = 1_2$

exponent: $10 + 127 = 137 = 1000\ 1001_2$,

mantissa (decimaldel): $1{,}357 = ,010\ 1101\ 1011\ 0010\ 0010\ 1101_2$

binär form:

$1100\ 0100\ 1010\ 1101\ 1011\ 0010\ 0010\ 1101_2$

Av dessa två tal kan endast 0,5 representeras exakt som ett binärt flyttal medan det andra värdet endast representeras av ett närmevärde (det relativa felet är ca $5 \cdot 10^{-9}$).

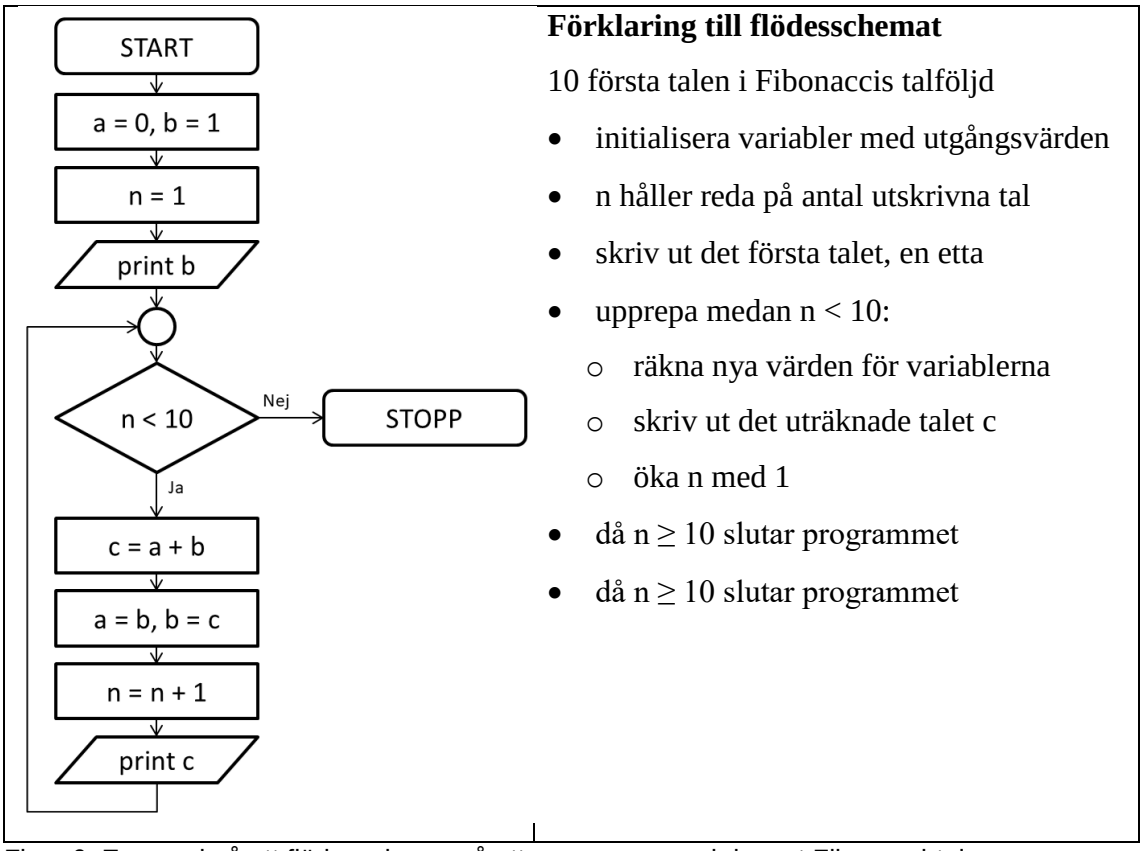
# Flödesschema

Flödesscheman är en visuell modell för en algoritm. Flödesscheman användes till en början för att planera mekaniska processer, men är också en mycket användbar metod för planering av algoritmer i programmering.

Det finns en hel massa med definierade symboler som kan användas i flödesscheman, här nämns endast de vanligaste för bruk i planering av ett dataprogram.

Tabell 17. Förteckning av de mest använda symbolerna i flödesscheman vid programmering.

| **Symbol** | **Namn** | **Funktion** |
|---|---|---|
|  | Start / stopp | Startpunkt eller stoppunkt för programmet |
|  | Process | Utför en funktion som t.ex. addition av variabler |
|  | Val | Selektion enligt en Boolesk logisk sats |
|  | Förbindelse | Används för att sammakoppla delar i ett flödesschema |
|  | In- / output | Läsa/skriva data från/till bl.a. tangentbord respektive skärm |



Figur 6. Exempel på ett flödesschema på ett program som skriver ut Fibonacci-tal.

# Olika typer av uppgifter

Eftersom inget undervisningsmaterial finns för den aktuella matematikkursen i gymnasiet, så tas här upp olika typer av uppgifter som egentligen främst är tänkta att fungera som grund för tankar till uppgifter som hanteras i kursen. Det är inte meningen att man skall behöva gå igenom alla dessa uppgifter med en undervisningsgrupp, utan välja lämpliga uppgifter som grund för egna uppgifter.

Uppgifterna som här tas upp är av varierande svårighetsgrad. I början är det främst uppgifter menade som repetition för hur de olika grundbegreppen i programmering fungerar. Därefter kommer det uppgiftstyper som direkt kopplar till kursens centrala innehåll med primtal och faktorisering. Skillnaden mellan rekursiva och iterativa funktioner tas upp i ett skilt underkapitel.

Numerisk lösning av ekvationer samt numerisk integration kan ses som mera avancerade uppgiftstyper då de kräver mera matematisk kunskap för att förstå. Men likväl är de exempel på fall där man i verkligheten använder programmering för problemlösning.

Även två enkla sorteringsmetoder hanteras främst av den orsaken att sortering som algoritm nämns i kursens centrala innehåll. De hanterade sorteringsrutinerna är inte speciellt svåra att förstå hur och varför de fungerar, men är inte rutiner som används i verkliga applikationer då deras tidskomplexitet är för stor.

Slutligen finns det en uppgift angående sannolikhet och en annan angående pseudo-slumptal som egentligen ligger utanför kursens innehåll, men är medtagna som exempel på lite mera krävande uppgifter för studeranden som redan kan de grundläggande sakerna och vill ha lite mera utmaningar i programmeringen.

De flesta exemplen baserar sig programmeringsteknik och matematik som kan anses allmän kunskap och därför hänvisas här inte i dessa fall till källor.

## Repetition av variabler, funktioner, villkorssatser och loopar

Några av de viktigaste koncepten inom programmering är variabler och funktioner samt villkorssatser och upprepande loopar. Även om eleverna borde vara bekanta vid dessa

koncept från grundskolan måste man ändå gå igenom dessa som en repetition före man kan börja med mera krävande programmeringsuppgifter.

Utgångsläget bör vara det att eleverna har endast minimala kunskaper inom programmering, då olika grundskolor har anordnat programmeringsdelen på olika sätt. Därtill dyker frågan om vilket programmeringsspråk som eleverna tidigare har bekantat sig med och vilket språk som nu används i undervisningen. För att även repetitionen skall gagna matematiken kan repetitionen med fördel vara programmering av funktioner som utför addition och multiplikation med två positiva heltal och därefter division. Alla programmeringsspråk kan utföra både addition och multiplikation med så väl heltal som med decimaltal i flyttalsrepresentation, men för att programmeringskoncepten skall framstå så klara som möjligt utförs dessa operationer här genom att dela in operationerna i mindre delar.

Addition innebär att man till det ena talet adderar 1 så många gånger som det andra talet anger. Problemet som därefter måste lösas är hur denna algorimiska tanke överförs i en form som datorn förstår, det finns många olika sätt att göra detta, här används en relativt enkel iterativ metod med hjälp av en WHILE-loop.

Kod 1. Exempel på addition.

```
// Funktion som tar in 2 argument och returnerar deras summa
// Argumenten lagras i variablerna a och b
// Variabeln sum innehåller i slutet värdet som returneras
function add(a, b) {
  var sum = a;        // summa-variabeln sum antar först värdet a
  var idx = b;        // eftersom algoritmen är iterativ behövs en index-variabel
  while(idx > 0) {    // upprepa medan idx är större än noll
    sum++;            // öka sum med 1
    idx--;            // minska idx med 1
  }
  return sum;         // returnera den uträknade summan i sum
}
```

Multiplikation innebär på motsvarande sätt att man adderar ihop det ena talet så många gånger som det andra talet anger, med utgångsvärdet noll. Koden för addition kunde med små omändringar även användas här, men här använder här vi istället en FOR-loop (initialisering av en FOR-loop görs ofta på en rad, men för att få kommentarerna i koden att vara så tydliga som möjligt är den här gjord på tre rader).

Kod 2. Exempel på multiplikation.

```
// Funktion som tar in 2 argument och returnerar deras produkt
// Argumenten lagras i variablerna n och a
// Variabeln sum innehåller i slutet värdet som returneras
function mult(n, a) {
  var sum = 0;         // variabeln sum antar först värdet 0
  for(var idx = 0;     // använd idx som indexerings-variabel
      idx < n;         // fortsätt så länge som idx är mindre än n
      idx++) {         // efter varje iteration ökas idx med 1
    sum += a;          // öka sum-variabelns värde med a
  }
  return sum;          // returnera den uträknade summan = produkten
}
```

Dessa två exempel borde vara relativt enkla att utföra, vill man öka svårighetsgraden en aning kan man ta till division med positiva heltal (reella tal kräver en mera komplicerad algoritm och kan därför inte tas som en repetitionsuppgift). I frågan om bråk har vi två olika divisioner som kan gås igenom, de är båda också mycket viktiga inom programmering. Den ena typen av division returnerar endast heltalsdelen och den andra endast restdelen.

Divisionsalgoritmen som returnerar heltalsdelen borde vara bekant för eleverna från grundskolan, alltså hur många gånger man kan subtrahera det ena talet från det andra förrän differensen blir icke-positiv. Denna algoritm är i det avseende annorlunda än algoritmerna för addition och multiplikation att här är det frågan om antalet gånger subtraktionen går att utföras som skall hållas reda på, inte ett värde som bestäms direkt från de givna värdena.

Kod 3. Exempel på heltalsdivision (DIV).

```
// Funktion som tar in 2 argument och returnerar heltalsdelen efter division
// Argumenten lagras i variablerna a och b
// Variabeln count innehåller i slutet värdet som returneras
// a DIV b
function div(a, b) {
  var count = 0;      // variabeln count räknar antal gånger subtratktionen utförs
  while(a > b) {      // WHILE-loopen fortsätter medan t är större än n
    a -= b;           // a:s värde minskas med b
    count++;          // count ökas med 1
  }
  return count;       // returnera antal gånger subtraktionen kunde utföras
}
```

Den andra divisionsalgoritmen, den som returnerar restdelen efter den utförda divisionen är mycket lika som den förra algoritmen, men denna gång behöver man inte hålla reda på hur många gånger subtraktionen går att utföra, utan istället är det endast det värde som återstår efter subtraktionerna i slutet som gäller.

Kod 4. Exempel på restdivision (MOD).

```
// Funktion som tar in 2 argument och returnerar heltalsdelen efter division
// Argumenten lagras i variablerna a och b
// Variabeln a innehåller i slutet värdet som returneras
// a MOD b
function mod(a, b) {
  while(a > b) {      // WHILE-loopen fortsätter medan a är större än b
    a -= b;           // a:s värde minskas med b
  }
  return a;           // returnera värdet som återstår efter alla subtraktioner
}
```

Dessa båda funktioner kan med fördel slås ihop till endast en funktion, men då måste två värden returneras på en gång. I JavaScript kan detta göras med en array eller ett objekt på följande sätt:

Kod 5. Exempel på division som returnerar både heltals- och restdelen.

```
function div_mod(a, b) {
  var count = 0;
  while(a > b) {
    a -= b;
    count++;
  }
  return [count, a];                    // returneras som en array
}

// användning:
dm = div_mod(100,9);
// heltalsdelen:  dm[0]
// restdelen:     dm[1]
```

Huruvida ovanstående exempel är funktioner man i verkligheten behöver göra själv i JavaScript är osäkert då motsvarande funktioner som redan nämnts är mycket viktiga inom programmering och därför också finns implementerade i själva språkets grundfunktioner. Med JavaScripts $parseInt()$-funktion får man heltalsdelen och med modulo-funktionen, som är gjord som en räkneoperation med procenttecknet, får man restdelen. Exempel: $parseInt(5/2) = 2$ och $5 \% 2 = 1$. Dessa två funktioner är bra att låta eleverna veta om, även om man inte tar hanterar exemplen ovan, då många matematiska algoritmer överskrivna till programmeringsspråk behöver dessa. Speciellt modulo-funktionen (eller %-operationen i JavaScript) kan vara extremt användbar till exempel då man skall göra en algoritm som hanterar de enskilda siffrorna i ett tal, med modulo kan man enkelt få ut siffran för en specifik talposition (ex: för att få ut tiotalssiffran i talet $x$ kan man ta $x \% 100 - x \% 10$).

## Primtal och -faktorisering

Eftersom primtal och primtalsfaktorisering är grundläggande kunskaper i matematiken och de också relativt enkelt kan överföras till programkod är detta ett ypperligt sätt att testa programmeringskunskaper. Här hanteras två metoder för att finna primtal, den traditionella ”Eratosthenes såll”-metoden och en metod baserat på den redan nämnda modulo-operationen.

I Eratosthenes såll är idén den att man ur ett bestämt intervall heltal, började från 2, iterativt går igenom varje kvarvarande tal genom att alltid välja det minsta kvarvarande talet som jämförelsetal. Vid varje iteration plockas alla tal som är delbart med jämförelsetalet bort. Alla använda jämförelsetal bildar mängden primtal i det hanterade intervallet.

För att kunna göra detta i JavaScript utan att göra ett onödigt långt och komplicerat program används här datastrukturen $Map$, som binder ihop en nyckel med ett värde. $Map$ möjliggör testning av en nyckels existens och även enkel borttagning av nyckel/värde-par.

Kod 6. Exempel på Eratosthenes såll för att hitta primtal.

```
function ErasPrimes(x) {
  var numbers = new Map();  // Map-datastruktur (för alla tal i intevallet [2, x])
  var primes = [];          // Variabel som populeras med primtal
  for(var i = 2;            // i som indexeringsvariabel med utgångsvärdet 2
      i <= x;               // upprepa FOR-loopen medan i är högst lika med x
      i++) {                // öka i:s värde med efter varje iteration
    numbers.set(i, true);   // skapa nyckel/värde-par med värdet true
  }
  for(var i = 2;            // iterera igenom alla heltal mellan 2 och x
      i <= x;               // i = 1, 3, 4, ... , x
      i++) {                //
    if(numbers.has(i)) {    // om nyckeln i existerar i numbers, så:
      primes.push(i);       //  nyckeln är ett primtal, spara detta
      for(var j = i;        //  iterera igenom alla multipler upptill x för i
          j <= x;           //   j = i, 2i, 3i, ..., x - x MOD i
          j += i) {         //
        numbers.delete(j);  //    ta bort nyckeln j från numbers
      }
    }
  }
  return primes;            // returnera en array med primtal i stigande ordning
}
```

Den andra metoden för att hitta primtal baserar sig på modulo-funktionen och blir därför mycket kortare, och använder därtill mycket lite datorminne, då endast de funna primtalen sparas. I Erastothenes såll måste en datastruktur som innehåller alla heltal från 2 upp till den översta gränsen för primtal sparas i en datastruktur. För datorer är metoden med modulo inget problem, men på Erastothenes tid för över 2000 år sedan

skulle denna metod vara mycket krävande med olika divisionsmetoder istället för endast addition som Erastothenes använde i sin metod.

Modulo-metoden är mycket simpel där man itererar igenom alla heltal från 2 upp till den sökta övre gränsen, därefter granskas om talet är en faktor i det sökta talet med modulo-operationen. Modulo-operationen upprepas så många gånger den aktuella faktorn finns i talet, och för varje fynd av en faktor divideras denna faktor bort.

Kod 7. Exempel på att söka primtal med hjälp av modulo-operatorn.

```
function getPrimes(x) {
  var primes = [];          // array som populeras med primtal
  for(var p = 2;            // iterera igenom alla heltal mellan 2 och x
      p <= x;               //
      p++) {                //
    while(x % p == 0) {     // medan x MOD p = 0, sök alla faktorer av samma värde
      primes.push(p);       // p är ett primtal om restdelen är 0, spara talet
      x /= p;               // dividera x med p, alltså dividera bort en faktor p
    }
  };
  return primes;            // returnera en array med primtalen i stigande ordning
}
```

Primtalsfaktorisering kan utnyttja valfritt någon av de ovanstående metoderna för att hitta primtal. Man söker först alla möjliga primtal mellan 2 och det tal som skall faktoriseras, intervalländpunkterna medräknade. Därefter itererar man igenom varje primtal och dividerar bort denna faktor så många gånger den existerar i talet och registrerar detta.

Kod 8. Exempel på primtalsfaktorisering.

```
// Funktion som tar in ett argument och returnerar talets primtalsfaktorer
// Argumentet lagras i variabeln x
// Variabeln factors innehåller i slutet en array av faktorer som returneras
function primfact(x) {
  var primes = getPrimes(x);    // sök alla dessa primtal, i fallande ordning
  factors = [];                 // variabel för de funna faktorerna
  primes.forEach(prime => {     // för varje primtal:
    while(x % prime == 0) {     //   medan primtalet är en faktor:
      factors.push(prime);      //     spara primtalsfaktorn
      x /= prime;               //     dividera x med denna primtalsfaktor
    }
  });
  return factors;               // returnera primtalsfaktorerna i stigande ordning
}
```

## Största Gemensamma Divisor (Euklides algoritm)

Att hitta den största gemensamma divisor (SGD) för två tal har länge sysselsatt matematiker, och även om metoden som här demonstreras kallas för Euklides algoritm finns det belägg för att den var känd redan före Euklides tid (Burton 2011). Det vore möjligt att söka SGD också genom att söka primtalsfaktorer i båda talen och från dessa bilda SGD, men Euklides algoritm skapar betydligt kortare kod.

Euklides algoritm bygger på vetskapen om att talet a alltid kan skrivas som $a = bq + r$, där $a$, $b$, $q$ och $r$ är heltal. Om man nu söker ett tal som delar både $a$ och $b$, måste detta samma tal även dela $r$. Nu kan man istället för att söka vilket tal som delar $a$ och $b$ söka ett tal som delar $b$ och $r$. Detta upprepar man så länge tills $b = r$, den största gemensamma faktorn för $a$ och $b$ är då $r$, vilket också gäller för de ursprungliga $a$ och $b$. (Rosen 2012)

Kod 9. Exempel på Största Gemensamma Divisor med Euklides algoritm.

```
// Funktion som tar in 2 argument och returnerar deras SGD
// Argumenten lagras i variablerna a och b
// Variabeln a innehåller i slutet värdet som returneras
function euclidesGCD(a, b) {
  while(a != b) {               // medan a olika b
    if(a > b) {                 // om a större än b:
      a -= b;                   //   subtrahera b från a
    } else {                    // annars:
      b -= a;                   //   subtrahera a från b
    }
  }
  return a;                     // returnera a
}
```

Eftersom algoritmen baserar sig på ekvationen $a = bq + r$ (kan läsas som: $b$ går $q$ gånger i $a$ och $r$ är resten), vilket egentligen betyder att $r = a\,MOD\,b$ (och $q = a\,DIV\,b$) kan funktionen göras mycket enklare.

Kod 10. Exempel på Största Gemensamma Divisor med hjälp av modulo-operatorn.

```
function GCD(a, b) {
  while(b != 0) {
    var r = a % b;
    a = b;
    b = r;
  }
  return a;
}
```

## Rekursiva funktioner

Rekursiva funktioner är relativt vanliga inom matematiken och de kan också programmeras som sådana utan att konvertera funktionen till en iterativ funktion. De flesta programmeringsspråken understöder rekursivitet, dock kan det finnas skillnader i hur djup eller komplicerad rekursiviteten kan vara då rekursion kan kräva mycket minne eftersom funktionens tillstånd måste sparas i minne före varje rekursion för att kunna återhämtas då rekursionen nystas tillbaka. I en del programmeringsspråk (ex. olika Lisp varianter) är rekursion att föredra framom iteration, men i de flesta moderna programmeringsspråk är iteration rekommenderad.

Till följande tas här upp två av de vanligaste exemplen för rekursion, fakultet och Fibonaccis talföljd, samt deras motsvarigheter som iterativa funktioner. Även om alla beräkningsbara rekursiva funktioner kan skrivas som iterativa funktioner är det inte alltid möjligt att bestämma denna iterativa funktion, då är den enda möjligheten att behålla rekursiviteten. Av denna orsak är det bra för studeranden att bekanta sig även med rekursiva funktioner inom programmering.

I frågan om rekursiva funktioner är det ytterst viktigt att komma ihåg att inkludera ett villkor för avslutande av den rekursiva funktionen. Detta är naturligtvis lika viktigt i frågan om iterativa funktioner, men då dessa använder sig av de inbyggda upprepningsmetoderna (FOR, WHILE mm.) kan de inte av bara misstag glömmas bort.

### Fakultet

I gymnasiematematiken behövs fakultet egentligen endast inom sannolikhetskalkyl då antal möjliga kombinationer och permutationer skall bestämmas, men är som övning för rekursiva funktioner ett ypperligt första steg.

Fakultet definieras som $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$, då $n > 0$, och $0! = 1$. Denna definition kan ses som rekursiv då $n! = n \cdot (n-1)!$. Iterativt är det enklare för nybörjare i programmering att hantera fakultet enligt $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-2) \cdot (n-1) \cdot n$, även om det är möjligt att göra loopar som räknar neråt.

Kod 11. Exempel på fakultet rekursivt.

```
function factRecur(x) {
  if(x == 0) {
    return 1;
  }
  return x*factRecur(x-1);
}
```

Kod 12. Exempel på fakultet iterativt.

```
function factIter(x) {
  var f = 1;
  for(var i = 1; i <= x; i++) {
    f *= i;
  }
  return f;
}
```

### Fibonaccis talföljd

Fibonaccis tal som dyker upp på flera ställen i matematiken definieras rekursivt som summan av de två föregående talen då utgångstalen är 0 och 1. Talföljden innehåller då följande tal: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, … och förhållandet mellan ett Fibonacci-tal och det föregående konvergerar mot det gyllene snittet. Fibonaccis talföljd är även enkel att hantera iterativt, då endast de två senaste värdena behövs hållas i minnet för varje iteration.

Den rekursiva funktionen för Fibonaccis-tal har utkommenterat ett kortare och aningen snabbare sätt att hantera avslutningsvillkorken för $n \in \{0, 1, 2\}$. Den aningen märkliga syntaxen ”$n == 0 ? 0 : 1$” är en så kallad ternär operation bestående av tre delar: ett villkor, värde då villkoret är sant och värde då villkoret är falskt. Därmed får denna ternära operation värdet 0 då $n$ är 0 och 1 i alla övriga fall.

Kod 13. Exempel på Fibonacci-tal rekursivt.

```
function fibRecur(n) {
  if(n == 0) {                  // om n = 1 eller n = 2:
    return 0;                   // returnera 1, rekursionen färdig
  }
  if(n == 1 || n == 2) {        // om n = 1 eller n = 2:
    return 1;                   // returnera 1, rekursionen färdig
  }
  // if(n < 3) {                // om n < 3 (n = 0, 1, 2):
  //   return n == 0 ? 0 : 1;   // returnera 0 om n = 0, annars 1, rek. färdig
  //}
  return fibRecur(n-1) +        // returnera F(n-1) + F(n-2)
         fibRecur(n-2);
}
```

I versionen med iteration hanteras fallet $n = 0$ som ett specialfall. Variabeln f innehåller det senast uträknade fibonacci talet och variablerna $a$ och $b$ de två föregående. I fall man vill få ut en lista på alla Fibonacci-tal upptill $n$ kan man skapa en array och sedan populera den med varje uträknat värde $f$.

Kod 14. Exempel på Fibonacci-tal iterativt.

```
function fibIter(n) {
  if(n == 0) {
    return 0;
  }
  var f = 1;                    // börja med värdet 1
  var a = 1;                    // a är "F(n-2)", först 1
  var b = 1;                    // b är "F(n-1)", först 1
  for(var i = 3; i <= n; i++) { // i = 3, 4, 5, ..., n-1, n
    f = a + b;                  // "F(n) = F(n-2) + F(n-1)"
    a = b;                      // "nästa F(n-2)" := "förra F(n-1)"
    b = f;                      // "nästa F(n-1)" := "förra F(n)"
  }
  return f;                     // returnera det n:te Fibonacci-talet
}
```

## Numerisk lösning av nollställen

Eftersom de flesta programmeringsspråken inte innehåller metoder för att lösa algebraiska uttryck, så måste man använda numeriska metoder för att lösa dessa. Därtill finns det algebraiska uttryck som inte kan lösas med vanliga lösningsmetoder, exempelvis en polynomfunktion av högre än fjärde graden. Alla dylika problem kan överföras att vara ett nollställesproblem om alla termer flyttas till det ena ledet och det andra ledet då blir 0.

Numeriska lösningsmetoder kan användas på valfria funktioner, men i detta sammanhang använder vi endast polynomfunktioner. Det finns bibliotek till JavaScript som kan laddas in i programkoden vilka understöder polynom, men eftersom man här endast behöver kunna bestämma polynomfunktionens värde och bestämma derivatafunktionen är här en kort kod som klarar av dessa uppgifter.

Kod 15. En klass för polynom med 3 metoder: värde, derivata och till strängformat.

```
function Polynom(...coefs) { // most .. least
  this.coefs = coefs.reverse(); // least .. most
  this.value = function(x) {
    var value = 0;
    for(var i = 0; i < this.coefs.length; i++) {
      value += coefs[i] * Math.pow(x, i);
    }
    return value;
  }
  this.derivative = function() {
    var newCoefs = [];
    for(var i = 1 ; i < this.coefs.length; i++) {
      newCoefs[i - 1] = coefs[i] * i;
    }
    return new Polynom(...newCoefs.reverse());
  }
  this.toString = function(x = "x") {
    var monoms = [];
    for(var i = 0; i < this.coefs.length; i++) {
      if(coefs[i] == 0) { continue; }
      var cTidy = coefs[i] == 1 ? "" : coefs[i];
      if(i == 0) { monoms.push(coefs[i]); }
      else if(i == 1) { monoms.push(cTidy + x); }
      else { monoms.push(cTidy + x + "^" + i); }
    }
    if(monoms.length == 0) { monoms.push(0); }
    return monoms.reverse().join(" + ");
  }
}
```

Programmet ovan är gjort enligt en äldre version av JavaScript och därför är Polynom inte egentligen en funktion utan en klass som kan instansieras. Koden utnyttjar flera metoder för att göra koden kortare, men eftersom denna kods uppbyggnad inte är det väsentliga här, så förklaras den inte här i detalj.

Kod 16. Exempel på hur Polynom-klassen används.

```
f = new Polynom(1, 2, 3, 4, 5);
d = f.derivative();
f.toString();                   // "x^4 + 2x^3 + 3x^2 + 4x + 5"
d.toString();                   // "4x^3 + 6x^2 + 6x + 4"
f.value(3);                     // "179"
d.value(4);                     // "380"
```

Polynom-klassen klarar inte av att hyfsa alla uttryck helt korrekt. Om en term har ett negativt förtecken och inte är uttryckets första term kommer termen att skrivas ut i formen ”$+ -a$”. Det vore inte svårt att få denna hyfsning att bli korrekt, men då skulle klassen bli längre och därtill är utskriftsmetoden för funktionen endast till för eventuell debuggning av kod som använder denna klass.

De numeriska metoderna baserar sig på Bolzanos sats, enligt vilken det för en funktion f som är kontinuerlig i intervallet $[a, b]$ och $f(a)$ har olika förtecken än $f(b)$ har funktionen också minst ett nollställe i intervallet $[a, b]$. Eftersom här endast används polynomfunktioner uppfylls villkoret för kontinuitet, och vi antar att det finns ett nollställe i intervallet $[a, b]$, men detta kontrolleras inte i koden. Detta innebär att programmet kan hamna i en oändlig loop, alltså att programmet aldrig avslutas. För att motarbeta detta problem kan man bestämma ett maximalt antal iterationer. Ett annat alternativ kunde vara att bestämma en maximal tid programmet får köras, men denna metod kan skapa oväntade effekter då användarens dators hastighet kan vara avgörande ifall en lösning hittas eller ej före programmet avslutas.

### Halveringsmetoden

Vi antar här att villkoren för Bolzanos sats uppfylls och att funktionen $f$ är kontinuerlig och har minst ett nollställe i intervallet $[a, b]$. Genom att för varje iteration bestämma intervallets mittpunkt $c$ och sedan välja den halvan av intervallet $[a, b]$ som delats med punkten $c$ som fortsättningsvis uppfyller Bolzanos sats. Därmed halveras intervallets storlek efter varje iteration. Algoritmen avslutas då intervalländpunkterna $a$ och $b$ ligger så nära varandra att deras differens är mindre än den valda noggrannheten.

Kod 17. Exempel på halveringsmetoden.

```
function halfIntervall(f, a, b, e = 0.000001) {
  while(Math.abs(b - a) > Math.abs(e)) { // medan noggrannheten ej uppnådd:
    var c = (b + a) / 2;                 //   beräkna mittpunken mellan a och b
    var sa = Math.sign(f.value(a));      //   förtecknet för a
    var sc = Math.sign(f.value(c));      //   förtecknet för b
    if( sc == sa ) {                     //   om c och a har samma förtecken:
      a = c;                             //     ändra den vänstra int.ändpunkten
    } else {                             //   i andra fall:
      b = c;                             //     ändra den högra int.ändpunkten
    }
  }
  return c;                              // returnera den sista räknade mittpunkten
}
```

Programmet använder den tidigare nämnda Polynom-klassen och Math.sign-funktionen för att bestämma förtecknet på ett uttryck, om det valda programmeringsspråket inte har en motsvarande funktion kan man bestämma förtecknet genom att dividera variabelns värde med dess absolutbelopp. En annan möjlighet är att bestämma förtecknet på variablernas produkt, om förtecknet för produkten är negativt var variablernas förtecken olika i andra fall lika. Funktionen antar värdet 0,000001 för felmarginalen, men detta kan justeras då funktionen kallas.

Kod 18. Exempel på använding av halveringsmetoden.

```
f = new Polynom(1, 1, -3);
fx0 = halfIntervall(f, 0, 2);              // "1.3027753829956055"

g = new Polynom(2, -4, 3, -9);
gx0 = halfIntervall(g, 1, 3);              // "2.2314958572387695"
gx1 = halfIntervall(g, 1, 3, 0.01);        // "2.2265625" med felmarginal 0,01
```

### Newtons iterationsmetod

Newtons iterationsmetod använder funktionen derivata för att snabbare hitta funktionens nollställen. Funktionen som används i Newtons metod måste uppfylla villkoren i Bolzanos sats, och man utgår från ett värde som borde vara relativt nära det nollställe man vill bestämma. Anta att funktionen $f(x)$ är kontinuerlig och har ett nollställe i intervallet $[a, b]$. Man väljer utgångsvärdet $x_0$ och därefter bildar man tangenten genom punkten $(x_0, f(x_0))$ och sedan söker tangentens nollställe $x_1$, vilket är

enkelt då tangenten är en linjär funktion. Därefter upprepas detta med det nya uträknade $x$-värdet.

Tangenten $t_n$ går genom punkten $(x_n, f(x_n))$ och har riktningskoefficienten $k = f'(x_n)$ och kan därmed skrivas i formen $y - f(x_n) = f'(x_n)(x - x_n)$. Nollstället för tangenten och samtidigt det nästa iterationsvärdet blir då $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.

Kod 19. Exempel på Newtons iterationsmetod.

```
function newton(f, x, e = 0.000001) {
  d = f.derivative();                          // skapa derivatafunktionen till f
  while(Math.abs(f.value(x)) > Math.abs(e)) { // medan nogrannheten ej uppnådd:
    x = x - f.value(x)/d.value(x);             //   räkna ut det följande värdet
  }
  return x;                                    // returnera det sista värdet
}
```

## Numerisk integration

För numerisk integration gäller i stora drag samma som för sökandet av nollställen eftersom det inte går att bestämma explicita primitiva funktioner till alla funktioner. Man kan använda Riemann-summor för att beräkna närmevärden för integraler, men här används två andra metoder, nämligen trapetsmetoden och Simpsons metod. Båda dessa metoder lämpar sig väl för att lösas numeriskt genom programmering.
Här används samma Polynom-klass som tidigare för att göra koden lättare att förstå.

### Trapetsmetoden

I trapetsmetoden delas integreringsområdet in i ett bestämt antal lika breda delintervall, och med hjälp av funktionens värden i delintervallens ändpunkter bildas trapets vars areor är enkla ett beräkna. Summan på alla dessa trapets areor blir då
$A = h(\frac{1}{2}y_0 + y_1 + y_2 + \ldots + y_{n-2} + y_{n-1} + \frac{1}{2}y_n)$, när antalet delintervall är $n$, delintervallens bredd $h = (b - a) / n$, $x_n = a + hn$ och $y_n = f(x_n)$.
Enligt formeln skall den första och sista termen, $y_0$ och $y_n$ multipliceras med $\frac{1}{2}$. Detta kan hanteras på två sätt i ett program. Endera sålunda att dessa två termer hanteras skilt och itereringen gör på intervallet $[a + h, b - h]$. Det andra sättet, vilket används här

behandlar dessa termer lika som alla övriga termer i serien och felet korrigeras i slutet före resultatet returneras.

Kod 20. Exempel på trapetsmetoden.

```
function trapetzoid(f, a, b, n) {
  var h = (b - a) / n;                // delintervallens bredd (trapetsens höjd)
  var sum = 0;                        // summan börjar från 0
  for(var x = a; x <= b; x += h) {    // iterera igenom alla delintervall
    sum += f.value(x);                //  addera funktionens värde i x till summan
  }
  sum -= (f.value(a) + f.value(b)) / 2;   // korrigera summan
  return h * sum;                         // returnera h * summan
}
```

### Simpsons metod

I Simpsons metod delas integreringsområdet in i ett bestämt antal delintervall. Därefter skapar man med hjälp av varje delintervalls mittpunkt och ändpunkter ett polynom av andra graden vars graf går genom dessa punkter och därefter approximerar man integralens värde genom att integrera varje andra grads polynom i sitt intervall. Lyckligtvis behöver man inte utföra dessa integreringar för varje delintervall, utan formeln för Simpsons metod efter förenkling av uttrycket blir $\frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \ldots + 4y_{n-1} + y_n)$, där $h = (b-a)/n$ och integreringsintervallet är $[a, b]$ samt $n$ antal delintervall för polynom-funktioner (märk att vi här inte kräver att $n$ är delbart med två då delintervallens mittpunkter hanteras i koden och gör de egentliga delintevallens antal till $2n$).

I koden används en hjälpvariabel i som används för att få koefficienternas värden som används i Simpsons metod. Koefficienten räknas ut med formeln $(2 \cdot (i\ MOD\ 2) + 2)$, vilket ger alternerande värdena 2 och 4, då $i = 0, 1, 2, 3, \ldots$ eftersom $i\ MOD\ 2$ alltid är 0 för jämna värden på $i$ och i övriga fall 1. Den första och sista koefficienten får båda sålunda värdet 2, vilket är fel då de borde vara 1. Detta fel korrigeras i koden på motsvarande sätt som i trapetsmetoden före resultatet returneras.

Kod 21. Exempel på Simpsons metod.

```
function simpson(f, a, b, n) {
  var n = 2 * n;                    // fördubbla delintervallens egentliga antal
  var h = (b - a) / n;              // delintervallens bredd
  var sum = 0;                      // summan börjar med värdet 0
  var i = 0;                        // en hjälpvariabel för koefficienten
  for(var x = a; x <= b; x += h) { // iterera genom alla delintervall
    coef = (2 * (i % 2) + 2);       //   koefficientens värde bestäms
    sum += coef * f.value(x);       //   addera koefficienten * f(x) till summan
    i++;                            //   öka i:s värde med 1
  }
  sum -= (f.value(a) + f.value(b))       // korrigera summan
  return h/3 * sum;
}
```

## Sortering

Sorteringsalgoritmer är mycket vanliga inom programmering och det har forskats mycket i olika typer av sortering för att kunna optimera dessa i fråga om både tid och användning av utrymme. Det finns tiotals kända sorteringsalgoritmer, men de flesta är inte särdeles enkla att förstå hur de egentligen fungerar. Därför tas här upp endast två av de enklare algoritmerna, ”selection sort” och ”bubble sort”.

### *Stora-O*

För att kunna jämföra algoritmers kvalitet har man skapat olika karakteristikor som kan användas för att värdesätta algoritmerna, en av de mest använda är den så kallade stora-O noteringen (”big-O”). Med stora-O kan man bestämma den relativa tidskoplexiteten eller utrymmesbehovet. Stora-O symboliserar en så kallad ”worst case”-situation, alltså sämsta möjliga läget, vilket betyder att algoritmerna i verkligheten kan vara bättre än vad stora-O indikerar. I bland kan exempelvis ”bubble sort” som teoretiskt är $O(n^2)$ faktiskt vara av tidskomplexiteten $O(n)$.

Det optimala vore att kunna sortera $n$ stycken element i lineär tid, detta markeras som $O(n)$, andra vanligt förekommande komplexiteter är logaritmiska $O(n\,lb\,n)$, polynomiala $O(n^k)$ där $k = 2,3,4 \ldots$ och exponentiella $O(2^n)$. Basen för logaritm- och exponentialuttrycken är godtycklig, men då det ofta är fråga om delning av mängder i 2 delar är basen 2 naturlig att använda. Vi går inte här djupare in på hur stora-O noteringen fungerar, utan belyser den med en tabell med exempelvärden istället.

Tabell 18. Exempel på olika stora-O komplexiteter.

| **Typ** | **Stora-O** | **n = 10** | **n = 20** | **n=100** |
|---|---|---|---|---|
| Lineär | $O(n)$ | 10 | 20 | 100 |
| Logaritmisk | $O(n\ lb\ n)$ | 33 | 43 | 664 |
| Kvadratisk | $O(n^2)$ | 100 | 400 | 10 000 |
| Kubisk | $O(n^3)$ | 1 000 | 8 000 | 1 000 000 |
| Exponentiell | $O(2^n)$ | 1 024 | 1 048 576 | $1{,}3 \cdot 10^{30}$ |

Ur tabellen framgår att det ytterst sällan i annat än övningssyfte, eller extremt små data set, lönar sig att använda algoritmer med stora-O av exponentiell karaktär. Då det sällan är möjligt att uppnå ett lineärt stora-O är det eftersträvansvärt att försöka få stora-O att förbli logaritmisk, men inte heller detta är alltid möjligt.

### Selection sort

I ”selection sort”, eller en typ av urvalssortering, går man igenom alla elementen och väljer ut elementet med det minsta värdet, därefter upprepas detta tills alla element är utplockade. I exempelkoden nedan placeras de utvalda elementen i början av listan.

Kod 22. Exempel på "Selection sort".

```
function selectionSort(a) {
  for(var i = 0; i < a.length; i++) {       // gå igenom alla element
    var min = i;                            //  aktuella elementet som det minsta
    for(var j = i + 1; j < a.length; j++) { //  gå igenom alla efterfölj. element
      if(a[j] < a[min]) {                   //   om akt. elementet mindre än minsta
        min = j;                            //   sätt aktuella till minsta
      }
    }
    var tmp = a[i];                         //  byt värden mellan ...
    a[i] = a[min];                          //  ... det aktuella och ...
    a[min] = tmp;                           //  ... det minsta
  }
}
```

### Bubble sort

I ”bubble sort” bubblar element i en lista med mindre vikt uppåt, alltså element med mindre värden flyttas stegvis närmare början av listan. Algoritmen kan implementeras på flera olika sätt, här tillämpas en metod som är enkel att förstå och följa med.

Kod 23. Exempel på "Bubble sort".

```
function bubbleSort(a) {
  for(var i = 0; i < a.length; i++) {       // gå igenom alla element
    for(var j = i + 1; j < a.length; j++) { //  gå igenom alla efterfölj. element
      if(a[j] < a[i]) {                     //   om värdet till "höger" mindre:
        var tmp = a[i];                     //    byt värden mellan ...
        a[i] = a[j];                        //    ... det "vänstra" och ...
        a[j] = tmp;                         //    ... det "högra" värdet
      }
    }
  }
}
```

## Sannolikhet

I fråga om sannolikhetsuppgifter kan vissa uppgifter lösas helt med dator och så kallad ”brute force”-metod där man helt enkelt går igenom alla möjligheter. Denna metod kan inte alltid användas då det totala antalet utfall blir för stort, eller då komplexiteten för att kontrollera gynnsamma utfall blir för hög.

### Monte Carlo

Däremot är den så kallade Monte Carlo-metoden för uträkning av sannolikheter mycket väl lämpad för programmeringsuppgifter. I denna metod väljer man slumpmässigt olika utfall och registrerar sedan summan på de gynnsamma utfallens antal och antal totalt testade utfall. Förhållandet mellan dessa ger sedan den ”empiriska” sannolikheten för händelsen i fråga.

Man kan exempelvis söka ett närmevärde för $\pi$ genom att slumpmässigt välja talpar inom intervallen $[-1,1]$ och anta talparen $(x, y)$ som uppfyller villkoret $x^2 + y^2 < 1$ som gynsamma utfall.

Kod 24. Exempel på Monte Carlo-metoden.

```
function montecarloPI(n) {
  for(var i = 0; i < n; i++) {          // utför n stycken försök
    var x = Math.random() * 2 - 1;      //   ett slumptal i intervallet [-1,1]
    var y = Math.random() * 2 - 1;      //   ett slumptal i intervallet [-1,1]
    var hits = 0;                       //   variabel för gynnsamma utfall
    if(x*x + y*y < 1) {                 //   om x^2 + y^2 < 1:
      hits++;                           //     öka gynnsamma utfallens antal med 1
    }
  }
  return hits / n;                      // returnera gynnsamma utfall / n
}
```

## Pseudo-slumptalsgenerator

I exemplet angående sannolikhet används slumptal. I normala förhållanden har en dator inte verkliga slumptal utan alla slumptalsfunktioner är deterministiska och därför kallas dessa tal allmänt för pseudo-slumptal.

Den mest använda metoden för slumptal inom programmering är baserad på linjär kongruens enligt formeln $x_{n+1} = (ax_n + c)\ MOD\ m$, där $x_0$,$a$, $c$ och $m$ är heltal och det gäller att $2 \leq a < m$, $0 \leq c < m$ samt $0 \leq x_0 < m$. Ofta väljs $c = 0$ eller $c = 1$, $m$ skall vara ett stort tal, för att algoritmen skall ge bra slumptal får $a$ inte vara för litet eller för stort och därför är det ett säkert val att välja $a$ att ha en siffra färre än $m$. (Rosen 2012, Sedgewick 1988)

De resulterande sluptalens fördelning kan sedan kontrolleras med ett $\chi^2$-test, men denna typ av test ligger utanför ramarna för denna text. Som en liten varning kan ändå nämnas att alla slumptalsgeneratorer inte är bra, och beroende på vilken användning slumptalen har bör algoritmen testas före bruk.

Kod 25. Exempel på Pseudo-slumptal.

```
var psdRnd = {                  // ett globalt objekt för all data
  seed : 1,                     // utgångsvärdet
  mult : 16807,                 // koefficienten som multipliceras varje iter.
  add  : 0,                     // termen som adderas varje iter.
  mod  : Math.pow(2,31) - 1,    // modulus (i detta fall ett Mersenne primtal)
  last : -1                     // senaste värdet, -1 iter. inte påbörjad
}
function pseudorandom() {
  // vid första iterationen används "seed" som utgångsvärde
  psdRnd.last = (psdRnd.last == -1) ? psdRnd.seed : psdRnd.last;
  // x_(n+1) = (a * x_n + c) MOD m
  psdRnd.last = (psdRnd.mult * psdRnd.last + psdRnd.add) % psdRnd.mod;
  // returnera det uträknade värdet dividerat med modulo-värdet
  // detta ger ett värde i det slutna intervallet [0,1].
  return psdRnd.last / psdRnd.mod;
}
```

# Källor

Balanskat, A. och Engelhardt, K. Computing our future – Computer programming and coding – Priorities, school curricula and initiatives across Europe. European Schoolnet, 2015.

Blume, G. och Schoen, H. Mathematical Problem-Solving Performance of Eighth-Grade Programmers and Nonprogrammers. Journal for Research in Mathematics Education, vol. 19, no. 2, pp 142-156, 1988.

Broley, L., Caron, F. och Saint-Aubin, Y. Levels of Programming in Mathematical Research and University Mathematics Education. International Journal of Research in Undergraduate Mathematics Education, no. 4, pp 38-55, 2018.

Burton, D. The history of mathematics: an introduction. McGraw-Hill. 2011.

Chabaya, R. och Sherwood, B. Computational physics in the introductory calculus-based course. American Journal of Physics, vol. 76, nos. 4 & 5, pp 307-313, 2008.

Clements, D. From exercises and tasks to problems and projects; Unique contributions of computers to innovative mathematics education. Journal of Mathematical Behavior, no. 19, pp 9-47, 2000.

Dijkstra, E. Selected Writings on Computing: A Personal Perspective. Springer-Verlag, 1982. Internet: https://www.cs.utexas.edu/users/EWD/ewd04xx/EWD498.PDF. Hämtad 23.10.2019.

Fey, J. Technology and mathematics education: A survey of recent developments and important problems. Educational Studies in Mathematics, vol. 20, no. 3, pp 237-272, 1989.

Github. The State of the Octoverse: top programming languages of 2018. Internet: https://github.blog/2018-11-15-state-of-the-octoverse-top-programming-languages/. Hämtad: 23.10.2019.

Gomes, A. och Mendes, A. Learning to program - difficulties and solutions. International Conference on Engineering Education, 2007.

Grandell, L., Peltomäki, M., Back, R-J. och Salakoski, Tapio. Why Complicate Things? Introducing Programming in High School Using Python. Conferences in Research and Practice in Information Technology Series, vol 52, Januari 2006. Internet: https://www.researchgate.net/publication/31596786_Why_Complicate_Things_Introducing_Programming_in_High_School_Using_Python. Hämtad 1.12.2019.

Halén, O. Datorer i Tekniska museets samlingar. 2001. Internet: http://digitalamodeller.se/daedalus/kapitel/Datorer%20i%20Tekniska%20museets%20samlingar.pdf. Hämtad: 23.10.2019.

Heintz, F., Färnqvist, T. och Thorén, J. Proceedings från 5:e Utvecklingskonferensen för Sveriges ingenjörsutbildningar, ss. 15-18. Uppsala, november 2015. Internet: https://www.it.uu.se/research/publications/reports/2016-002/2016-002.pdf. Hämtat 1,12,2019.

Hellas, A. Retention in Introductory Programming. Unigrafia. Helsinki. 2017.

IEEE. IEEE Standard for Floating-Point Arithmetic, IEEE Std 754™-2008. The Institute of Electrical and Electronics Engineers, New York. 2008.

Ihantola, P. Automated Assessment of Programming Assignments. Aalto University publication series, Doctoral Dissertations 131/2011.

Jacobsson, R-M. och Melander A. Att införa programmering i matematik på gymnasiet. Examensarbete, Malmö Universitet. 2018.

Kivelä, S. Matematiikan opetus ja tietokoneaikakausi, 2004. Internet: http://matta.hut.fi/matta2/artikkelit/mathcomp04.pdf. Hämtad: 23.10.2019.

Lye, S. and Koh, J. Review on teaching and learning of computational thinking through programming: What is next for K-12? Computers in Human Behavior, no. 41, pp 51-61, 2014.

Sedgewick, R. Algorithms, Second edition. Addison-Wesley Publishing Company. New York. 1988.

Starr, W., Manaris, B. och Stavley, R. Bloom's Taxonomy Revisited: Specifying Assessable Learning Objectives in Computer Science. ACM SIGCSE Bulletin, Januari 2008.

Suominen, K. Kehittämistutkimus: Matematiikan aineenopettajaopiskelijoiden ohjelmointikoulutus. Pro gradu, Helsingin yliopisto, 2019.

Rosen, K. Discrete Mathematics and its Applications, Global Edition. McGraw-Hill Professional. 2012.

The Royal Society. Shut down or restart? The way forward for computing in UK schools. 2012.

Utbildningsstyrelsen. GLP2021– Utkast till grunderna för läroplanen för gymnasieutbildning 2019. Internet: https://www.oph.fi/sv/glp2021-grunderna-gymnasiets-laroplan-fornyas. Hämtad 23.10.2019.

Wiatrowski, C. och House, C. Logic Circuits and Microcomputer Systems, 6th printing. McGraw-Hill. 1988.

Wing, J. Computational thinking and thinking about computing. Philosophical Transactions of the Royal Society A, vol. 366, pp 3717-3725, 2008.